10-27-1994

High Level Preprocessor of a VHDL-based Design System

Karthikeyan Palanisamy
Portland State University

THESIS APPROVAL

The abstract and thesis of Karthikeyan Palanisamy for the Master of Science in Electrical and Computer Engineering were presented October 27, 1994, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS: [redacted]

Marek A. Perkowski, Chair

[redacted]

Malgorzata E. Chrzanowska-Jeske

[redacted]

Bradford R. Crain
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL: [redacted]

Rolf Schaumann, Chair
Department of Electrical Engineering

* *

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 10 January 1995

ABSTRACT

An abstract of the thesis of Karthikeyan Palanisamy for the Master of Science in Electrical and Computer Engineering presented October 27, 1994.

Title:HIGH LEVEL PREPROCESSOR OF A VHDL-BASED DESIGN SYSTEM

This thesis presents the work done on a design automation system in which high-level synthesis is integrated with logic synthesis. **DIADES,** is a design automation system developed at PSU, starts the synthesis process from a language called **ADL.** The major part of this thesis deals with transforming the **ADL** -based **DIADES** system into a **VHDL** -based **DIADES** system. In this thesis I have upgraded and modified the existing **DIADES** system so that it becomes a preprocessor to a comprehensive **VHDL** -based design system from Mentor Graphics.

The high-level synthesis in the **DIADES** system includes two stages: data path synthesis and control unit synthesis. The conversion of data path synthesis is done in this thesis. In the **DIADES** system a digital system is described on the behavioral level in terms of variables and operations using the language **ADL.** The digital system described in **ADL** is compiled to a format called **GRAPH** language. In the **GRAPH** language the behavior of a digital system is represented by a specific sequence of program statements. The descriptions in the **GRAPH** language is compiled to a format called **STRUCT** language. The system is described in the **STRUCT** language in terms of lists of nodes and arrows. The main task of this thesis is to convert the descriptions in the **GRAPH** language and the descriptions in the **STRUCT**

language to the **VHDL** format. All the generated **VHDL** Code will be Mentor Graphics **VHDL** format compatible, and all the **VHDL** code can be compiled, simulated and synthesised by the Mentor Graphics tools.

HIGH LEVEL PREPROCESSOR OF A VHDL-BASED DESIGN SYSTEM

by

KARTHIKEYAN PALANISAMY

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1994

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In recent years hardware description languages have been playing an important role in the synthesis of hardware. The VLSI technology with which modern digital systems are designed has advanced at such a tremendous pace within the past few years that the engineer is being outstripped of his ability to design complex state-of-the-art systems. Initially, various CAD tools (circuit analysis, logic minimization, layout, etc.) where made available to aid the designer in concurring such increasingly more difficult design tasks. Technology has reached a level, however, that calls for the realization of design automation system in which high-level synthesis is integrated with logic synthesis. **DIADES** design automation system which starts the synthesis process from a language called **ADL** (Algorthmic Description Language) has been developed at PSU in recent years. The main task of this thesis is to upgrade, modify and link the **DIADES** system so that it will become a pre-processor of an commercial **VHDL** -based synthesis/simulation environment from Mentor Graphics Corporation.

1.1 DIADES

The **DIADES** design automation system is a set of programs for the synthesis of digital circuits from high-level, behavioral descriptions. A digital system is described on the behavioral level in terms of variables and operations using a language called **ADL.** The behavioral description is compiled to a structural description, which is

composed of specific hardware units such as adders, buffers, and multiplexors. **ADL** is more like a programming language. It allows for easy description of parallel and sequential behavior of hardware. In this respect it is much closer to such specification languages as C or PASCAL. The main advantage is that it is easier to express parallelism in **ADL** than in standard HDL languages such as **VHDL** or **VERILOG.**

A digital system can be described at three levels, behavioral, functional, and structural. The language **ADL** is used for behavioral representation. **ADL** is a block-structured, algorithmic language developed especially for **DIADES** system. So it has some properties specific to **DIADES** design routines and methodologies. "High-level synthesis" in this thesis refers to the process of generating a digital circuit description on the structural level from a behavioral level description. The High-Level Synthesis in DIADES includes two stages: Data Path Synthesis and Control Unit Design. The data path unit contains the descriptions of the hardware elements needed to execute the behavioral algorithm.

The control unit controls the flow of data in the data path unit by producing register load signals, multiplexor addresses and other control signals. The control unit can be implemented as a micro-programmed unit or as a finite state machine using a PLA or another standard kind of logic.

Transformations and optimizations can be applied to the digital system description at all levels. Techniques from optimizing compilers, Boolean minimization, and Finite State machine minimization are some of the techniques that have been be applied in **DIADES.**

The goal of **DIADES** is to automate the process of digital system design. It is a powerful tool with many innovative and unique features. It can be used to design

dedicated micro-controllers, digital signal processors, and other types of systems. A block diagram of the current **DIADES** system is shown in Figure 1.

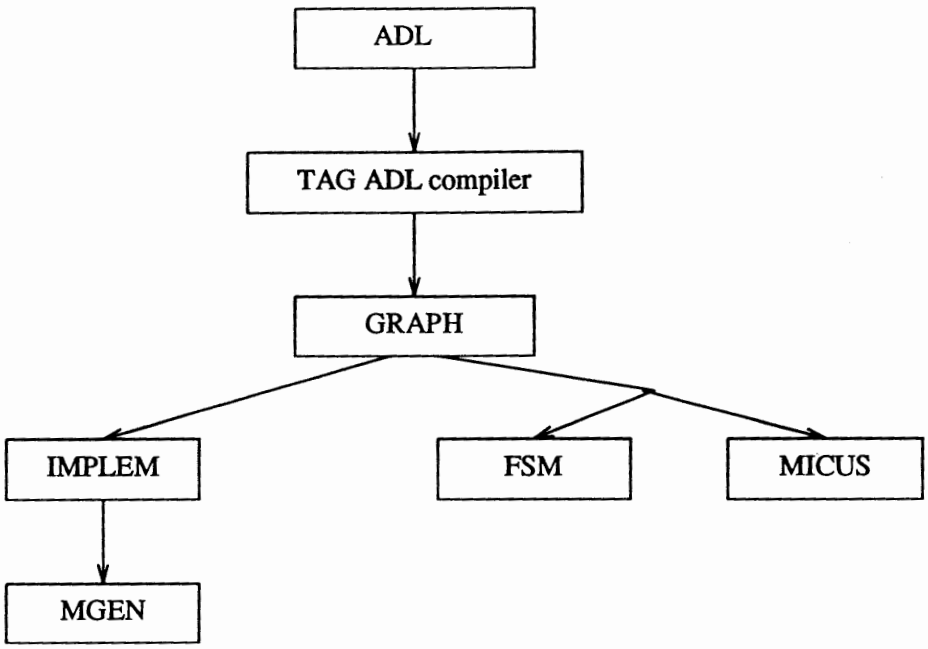

Figure 1. DIADES system.

In Figure 1, **TAG** is the **ADL** compiler. **GRAPH** is the main intermediate format of **DIADES.** The data path unit is performed by **IMPLEM** and **MGEN. IMPLEM** takes the descriptions in the form of **GRAPH** format as an input and generates a description in an abstract netlist language called **STRUCT. MGEN** takes **STRUCT** description as input and generates a netlist in an netlist format called **M** language. The control unit design is created by programs **FSM** and **MICUS. FSM** is the finite state machine control unit synthesizer, it takes the descriptions in the form of **GRAPH** format as input and generates a truth table data format for the combinational part of the finite state machine. The **MICUS** is the microprogram control unit design program that takes the descriptions in the form of **GRAPH** format as input and

generates the truth table of the control memory as its output.

1.2 VHDL-BASED DIADES

In this thesis we modify the existing **DIADES** system so that it will become a **VHDL** -based system. First the intermediate behavioral format of the **DIADES's** system **GRAPH** is converted to a **VHDL** code. Then the descriptions in the form of **STRUCT** format is converted to **VHDL** code. The block diagram of the new **VHDL** -linked **DIADES** system is shown in Figure 2.

All the generated **VHDL** code is in the Mentor Graphics **VHDL** format. The **VHDL** code generated by the **VHDL** -based **DIADES** system can be compiled by the **VHDL** -based mentor tools. The compiled file can be simulated or can be used for implementing it in the **FPGAs.** If the **FPGA** implementation option is chosen then the code should be synthesised and it fed to the Xilinx **FPGA** tools for technology maping. Then placement and routing can be done and then it can be implemented in the **FPGA** system. The **FPGA** option was given because of its almost zero turn-around time and manafacturing cost. There are also a lot of commercially available **FPGAs** in the market. The Xilinx **FPGA** is a RAM-based **FPGA** used by our group.

The Mentor Graphics system was selected because of its availabilty in the school and because of its wide range acceptance in the industry. The Mentor Graphics tools have many sophisticated Logic Synthesis, physical design, simulation or FPGA related programs that are absent from **DIADES. VHDL** was chosen because of its wide range and increasing use in industry. **VHDL** is a full form hardware description language. It is a very good language for detailed hardware specification. " The results of recent industry surveys, which indicate an increased project growth in the use of

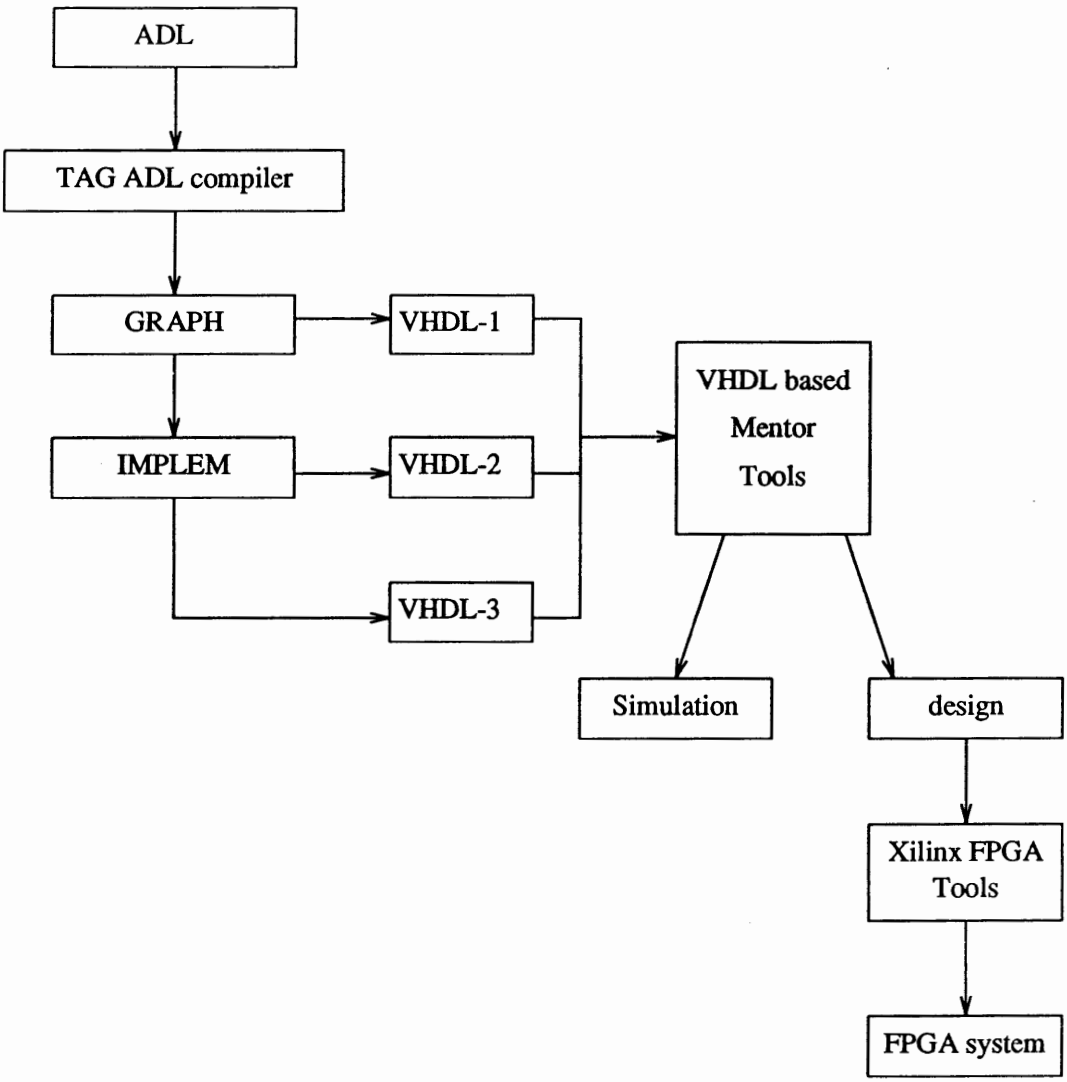

Figure 2. VHDL linked DIADES system.

VHDL, supporting the claim that **VHDL** is entering its deployment phase. In one survey 91% of the respondents presently not using **VHDL** plan to use the language in future [1]. In an informal survey the respondents projected an eightfold increase in the number of hardware designs using **VHDL** over the next two years [2].

The goal of this thesis is to first modify and improve the **DIADES** system and also to make it an **VHDL** -based system. The improving and modification for the

datapath unit of the **DIADES** system has been done in this thesis and the improvement and modification of the control unit is left for the future research.

CHAPTER II

DIADES DESIGN AUTOMATION SYSTEM

2.1 INTRODUCTION

The **DIADES** design automation system is a set of programs for the synthesis of digital circuits from high-level, behavioral descriptions [4]. The High-Level Synthesis in **DIADES** includes two stages: Data Path Synthesis and Control Unit Design. The data path synthesis and control unit synthesis start from a parallel **programGRAPH,** the form of description that includes both the control flow and the data flow graph. While the control unit is designed to be composed of either a microprogrammed unit, or Finite State machines. The Finite State machines are minimized in two dimensions (states and inputs), assigned and realized in logic. Several logic synthesis procedures, respective to various design styles and methodologies, can be used to design combinational parts of state machines, microprogrammed units and data path logic.

The **DIADES** system uses the language **ADL** for the behavioral description of the digital system. **ADL** is a language used to describe a circuit on a high level. A circuit's behavior can be described without knowing the basic elements of the circuit, or the internal states of the controlling state machine. The description is done in terms of arithmetic and logic operations. Control flow is described using *if_then_else, go,* and *while* statements [5]. A block diagram of the current **DIADES** system is shown in Figure 3.

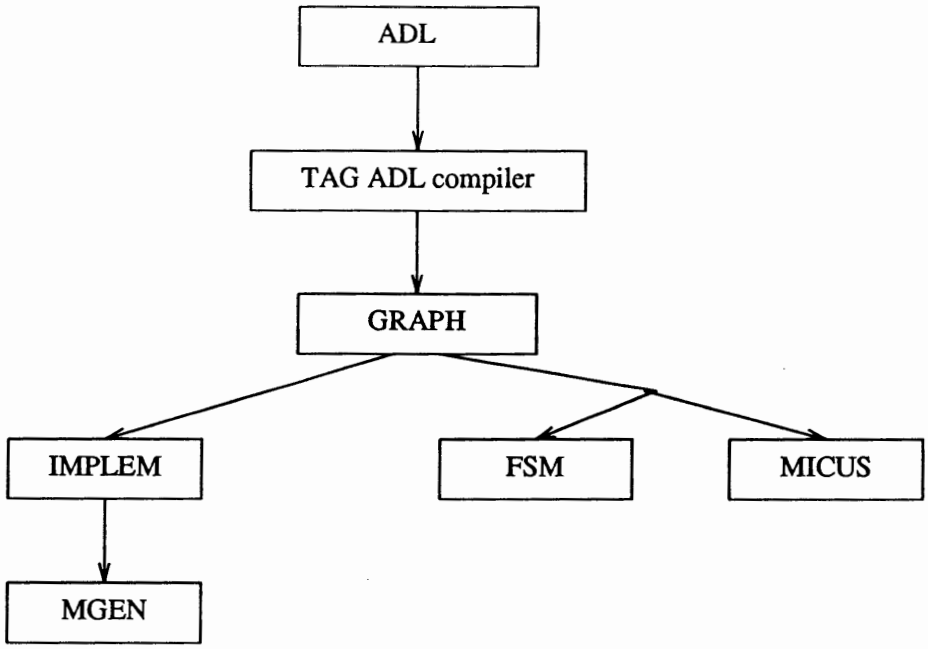

Figure 3. Current DIADES system.

The system is organized into four main sections. The first section takes a digital system description written in **ADL** and compiles it to a format called **GRAPH.** The descriptions in the **GRAPH** format represents the digital system at the behavioral level but in an easier to manipulate form. Transformations and high level optimizations such as decomposition of expressions and macro expansions take place upon the descriptions in the **GRAPH** format.

The descriptions in the **GRAPH** format is used by the data path and control unit generators. Further transformations and optimizations are applied to the digital system. Some changes in the data path are fed back into the control unit generator. The control unit generator selects one of two design styles, finite state machines and micro-programmed control units.

The fourth section takes the output of the two generator sections. This output,

which is an abstract structural form, is translated to a specific structural netlist format. Further transformations and optimizations can be applied at this level.

2.2 DATA PATH SYNTHESIS

First in **DIADES** the circuit descriptions in **ADL** is converted into **GRAPH** format, which is the main intermediate format for **DIADES.** After the descriptive format has been converted to **GRAPH** format the operations in the **GRAPH** format are scheduled for execution in specific control steps. The scheduling tool in **DIADES** tries to make the best use of resources, and to increase the concurrency of the descriptions in **GRAPH** format. The descriptions in **GRAPH** format is analyzed and the order of execution of operations in it is determined. Those operations that can be executed at the same time are identified. The scheduler looks for the best sequence of operations, resulting in the shortest overall control program. After scheduling, **DIADES** assigns variables to registers and operations to specific functional units.

After allocation, the design process splits into two separate processes. The data path unit is generated by the program **IMPLEM** and then by the program **MGEN.** The **DIADES** data path generator, **IMPLEM** , takes the descriptions in the form of **GRAPH** format as the input. The descriptions in the **GRAPH** format is mapped to a hardware structure in the abstract netlist language **STRUCT.** The functional units, registers, and input/output ports are represented by nodes, while the connections are represented by arrows. The **STRUCT** description is detailized by the program **MGEN,** which means that abstract descriptions of structural blocks are replaced with lower level descriptions of blocks and their connections. The output format is in

netlist format called **M** language. For instance, the **STRUCT** description specifies the "abstract block" BLOCK1 - COUNTER modulo 5 with loading input A, and clock CL. The equivalent **M** language description will specify all flip-flops and NAND gates of this counter and how they are connected, with an accuracy to each wire and gate input.

2.3 CONTROL UNIT SYNTHESIS

Two design styles are used in **DIADES** to design the control unit: the Microprogrammed Units and the Finite State Machines (FSMs). Both of them are designed starting from the descriptions in the **GRAPH** format, the main internal data representation language in **DIADES.** The result of the microprogram control unit generation is the microcode for control memory. The result of **FSM** control unit generation is the truth table or the Eqn (a set of Boolean equations) data formats for the combinational part of the **FSM.** The **FSM** synthesizer is composed of two stages, the **FSM** generation stage and **FSM** optimization stage. The first stage is a process to analyze the control flow **GRAPH** and assign machine states to the specified operations. It also encodes the input and the output signals for **FSM** structure between the control unit and the data path from different variants of the design scheme. The **FSM** optimization is composed of the state minimization program and the state assignment program. The state minimization reduces both the number of rows and the number of colums of state table, which is a unique feature of **DIADES.** The state assignment program encodes machine states to simplify the hardware implementation of the **FSM.**

The microprogrammed unit synthesizer, **MICUS,** first generates the SIMC (symbolic intermediate microcode) language description. Microinstruction compaction and optimization are necessary for efficient microcode. To translate **SIMC** further into object microcode a microassembler is used. Ultimately, a **TT** formatted truth table of control memory for logic minimization is generated. The optimization of the microcode is not performed separately in the control unit synthesis system, but is incorporated in the comprehensive process of data path scheduling and allocation and control unit design [6],[7].

The **DIADES** system is available in the PSU Sun network system. It is in the directory */stash/polo/newstuff/diades*. The **DIADES** system is made up of several programs written in Lisp, C, and Fortran. These programs are in a hierarchical directory structure in directory of */stash/polo*. **DIADES** is controlled by an interactive shell script called diades in the */stash/polo/newstuff/diades* directory. Each program can be executed seperately or the whole system can be executed automatically. When the shell script is executed, the user is presented a menu showing all commands and the **DIADES** framework.

CHAPTER III

ADL HARDWARE DESCRIPTION LANGUAGE

3.1 INTRODUCTION

ADL is an algorithmic language devoted mainly to describe the behavior of a digital system [8],[9]. An **ADL** program implements some algorithm. An algorithm is a step by step procedure for solving a problem or accomplishing some task. The procedure is organized into a behavioral series of assignment statements and control flow statments. **ADL** also describes systems at a *functional* level or *structural* level. Systems can be formulated using a mixed description on all three levels.

An **ADL** program is specified by a set of input and output ports, internal or intermediate variables, and the algorithm.

ADL Program: ((Input Ports
Output Ports
Internal Variables)
(Algorithm))

The algorithmic description is what sets this system apart from a functional level system. An **ADL** algorithm is similar to an algorithm in a programming language. The syntax is different but the semantics is quite close. The main difference is that a program algorithm is executed on a general purpose computer. The **DIADES** system generates a hardware processor executing only that specific algorithm.

Example 3.1.1 - An ADL program.

The header sections have been left out. The exact syntax is explained later. The program represents a simple system which adds two numbers and outputs the sum. If the result is equal to 10, the second number is subtracted from the first and the minuend is send to the output. This task is continuously repeated.

```
Input variables:   operand1, operand2 - 8 bit numbers.
Output variables:  answer - 8 bit number
Internal variable:  temp - holds answer
line 1: (((adl an example_circuit ....
line 2:              (input (op1 (p k1 8)) (op2 (p k1 8)) )
line 3:              (intern (temp (p k1 8)))
line 4:              (output (answer (p k1 8)))
line 5:    ((start) a
line 6:        10  (temp := (op1 + op2))
line 7:       (if (temp = 10) then (temp := (op1 - op2)))
line 8:       (answer := temp)
line 9:       (go 10)
line 10: )))
line 11: end
```

This system requires 3 clock cycles to operate. Statements are explained below:

Line 1 declares that this is a description in **ADL** language and the system name is "example circuit". The single letter "a" is a unique identifying symbol for the algorithm. Subroutines, if any, have different identifying symbols.

Lines 2 through 4 are the declarations. The declarations are similar to variable declarations in other languages. The only difference is that the functionality of

variables is defined at the start and does not change later in the program, i.e., input variables are used only for input.

Line 5 controls the start of the task. This statement is always executed when the system is powered up. However, this statement does not do anything and it is only used to initiate the operation of the control unit.

Line 6 is the first statement executed. It has the label *"10"*. Labels are always numbers. This is a simple arithmetic operation of addition. The result of the addition is stored in variable *temp*.

Line 7 is a conditional statement. If the relation is true, then statements after the *"then"* keyword are executed. If the relation is false, then the next statement is executed at line 7. The value in *temp* is compared with the value 10. If they are equal, then the subtraction is performed and the value stored in *temp*. There is no *"else"* keyword in this statement, although they are legal.

Line 8 is a simple assignment statement. The value *temp* is stored in the variable answer.

Line 9 is a *go* statement. This is the simplest control flow statement. The system will go back up to the statement in line 5 and execute the task again. New data is assumed to be at the input.

Lines 10 and 11 mark the end of this simple program. More complicated programs may have other lists and program instructions before the last right parenthesis. The *"end"* keyword is always at the very end of all programs.

As it can be seen, the syntax of the **ADL** has many parentheses and reminds that of Lisp. This is because **ADL** is an experimental language and we want to make it easily expandable.

3.2 PROGRAM STRUCTURE

The **ADL** language describing a digital system has four sections. The first section contains a few lines of parameters for the **ADL** compiler. The second section lists variables used in the program. Variables represent the inputs and outputs of a system as well as internal registers or memories. The third section contains the algorithm. The fourth section contains any subroutine descriptions.

The simple **ADL** program is first broken down into 2 lists. A declaration list and an algorithm list.

```
(( ( Declarations )
       ( Algorithm   )
))
```

The declaration list is composed of several lists.

```
(      program name
       ( Input variables    )
       ( Internal variables )
       ( Output variables   )
                    .
                    .
       ( Other Declarations )
)
```

The algorithm section is composed of statements. Each statement is a list.

```
( ( statement 1 )
```

(statement 2)

.

.

(statement n)

)

3.3 Declarations

Formal grammar of declarations

Declarations ::= (adl <symbol> <program name>

<clock list>

<input variable list>

<internal variable list>

<output variable list>

<subroutine list>

<constant list>

<identity list>

<symb list>

) ;

<clock list> ::= ((clock (<natural number>))) ;

<input variable list> ::= (input <variable declaration>+ / empty) ;

<internal variable list> ::= (intern <variable declaration>+ / empty) ;

<output variable list> ::= (output <variable declaration>+ / empty) ;

<variable declaration> ::= (<description> (<type>)) ;

<description> ::== <name> / <indexed variable> ;

<indexed variable> ::= (<name> [<size>]) ;

<size> ::= <natural number> / <index variable> ;

<type> ::= <p variable> / <d variable> ;

<p variable> ::= (p k1 <number of bits>) ;

<number of bits> ::= <positive integer> ;

<d variable> ::= (d) ;

<subroutine list> ::= (subr <subroutine element>+ / empty)

<subroutine element> ::= (<subroutine type> <symbol> <name> (<fix flag> <parameter>+)) ;

<subroutine type> ::= macro / block / logmacro / logblock ;

<fix flag> ::= fix / empty ;

<parameter> ::= <number> / <vector> / <parameter name> ;

<parameter name> ::= <name> ;

<constant list> ::= (const <list of parameters> / empty) ;

<identity list> ::= (iden <identity> / empty) ;*

<identity> ::= (<name> <expression>) ;

<name> ::= Any combination of letters and numbers ;

<expression> := <constant> / <variable> / <arithmetic expression>

/ <logical expression> / <predicate>

/ <expression> @ <expression> ;

3.4 VARIABLES

Variables in **ADL** are used to store data. There are 3 variable functions, input, output and internal. Input variables are used to input data to the system. Internal variables store data inside the system. Output variables are used to output data from the system. When the system is finally realized, variables are mapped to registers or memory. Because the **DIADES** system optimizes the final system's hardware there is not necessarily a one-to-one mapping of variables to registers.

There are two types of variable data. Parallel variables are vectors of binary ordered bits. Values of parallel variables are non-negative integers. Two's complement negation is done automatically in a statement by placing a minus sign in front of a variable. These are generally used for numbers or data. Logical variables are single bit variables and have two states, 1 and 0. These are usually used for flags and signals.

Example 3.4.1 - Parallel and Logical Variables.

"a" and "b" are declared as parallel variables. "c" is declared as a logical variable. To the right is a physical implementation.

```
(intern (a (p k1 8))
        (b (p k1 8))
        (c (d))     )
```

```
1          (b := (and a b)
                .
                .
                .
                .
2          (b := (and a c)
```

in statement 1, data from "a" and "b" is ANDed in parallel and stored in "b". In statement 2, the single bit from "c" is projected onto "a" and stored in "b".

Each variable can be treated as an array of bits. The array of bits is called a bit field. 1 bit binary values can be stored to or accessed from individual positions in a variable. 1 bit binary values consist of 1s and 0s. True and false symbols are not available but can be declared as constant 1s and 0s.

A range of bits in a variable can also be accessed. A range of bits is treated as a decimal number. The maximum value of a range of bits is:

2^(length of range)

Example 3.4.2 - Bit Fields and Ranges in Variables

(intern (a (p k1 8))) - 8 bit variable, values range from 0 to 255.

(a [0 % 0]) - 1st bit in a.

(a [2 % 3]) - bits 2 and 3 of a.

(intern (b (p k1 16))) - 16 bit variable, values range from 0 to 65535.

(b [14 % 14]) - 14th bit in b.

Array variables are arrays of simple variables. Array variables can only be internal types.

Example 3.4.3 - ADL arrays

(intern ((a [256]) (p ram 32))

(b (p k1 32)))

.

.

((a [34]) := 8)

(b := (a [34]))

((a [b]) := 10)

An indexed array variable has the following format for internal variables:

a) *((<name> [<size>]) (p <code> <bits>))*

This form is for array variables with the elements being parallel data.

OR

b) *((<name> [<size>]) (d))*

This form is for array variables with the elements being logical data. The code specification has a different meaning in array variable declarations. The **DIADES** user can specify how the array is to be implemented. There are two ways;

ram - Random Access Memory

k1 - register file

If the RAM option is selected, the array will be implemented as a standard RAM element with memory size and word size as specified. The user doesn't need to be concerned with any details such as addressing and read/write signals.

If the k1 option is selected, the array is implemented as an array of individual registers. Each register can be accessed independently of the others. This option results in a much larger area needed for the array variable.

Example 3.4.4 - Variable declarations

(a (p k1 32)) - 32 bit binary variable with name "a".

(b (p k1 6)) - 6 bit binary variable with name "b".

(c (d)) - single bit logical variable with name "c".

((d [32]) (p ram 8)) - array with 32 words. Each word is 8 bits.

Variable names start with a letter and can be any combination of letters and numbers. The length of the name is unlimited. In the main program, input variables are always implemented as input ports. They are always simple variables. They can not be assigned values from within the system and new values are determined from outside the system. These values can change at any moment of time. However the input ports are implemented as clocked buffers so new data is available each cycle.

Input variables in subroutines receive data from the main program. Depending on the subroutine type, the variables are implemented as registers or refer to variables in the main program. Internal variables contain values inside of the program, (or digital system). Data can be assigned to or retrieved, from these variables. Indexed variables are one dimensional arrays of simple variables. Normally these are specified as RAM in an **ADL** program. When the system is realized, internal variables may be mapped to registers or may not exist at all.

Example 3.4.5 - ADL Variables:

(intern (a (p k1 32))

(k (p k1 8))

((b [256]) (p ram 32)))

.

.

(a := (a + 1))

((b [23]) := (a - 5))

(a := (b [k]))

Output variables are used to output the data from the system. Unless an output variable is also declared as an intern variable, data cannot be retrieved from it. Depending on how the program is written, an output variable can either be a register or wired connection to some system element. If feedback into the system from an output variable is required, the variable is declared in both the intern and output declaration lists. The name, type and size are identical.

Example 3.4.6 - ADL output

(intern (a (p k1 32)))

(output (b (p k1 32)))

.

.

(b := a)

Example 3.4.7 - Intern and Output Variables

This example shows how output variables are also declared internal, so the data from the output is also available to the system.

(intern (a (p k1 16)) (b (p k1 16)))

(output (c (p k1 8)) (b (p k1 16)))

.

.

(b := 5)

(c := b)

3.5 CONSTANTS

Any number in **ADL** can be given a name. Once a constant is declared, the name can be used in any expression where the number can be used. Numbers in **ADL** are decimal only. A constant can be either a number or an array of numbers. If an array constant is declared, all array elements must be given values. The format for constant declarations is:

(const <constant elements>+)

<constant elements> ::= (<name> <number>)

Example 3.5.1 - ADL Constants:

(const (coeff_1 42)

(coeff_2 9))

3.6 IDENTITIES

An **identity** is a symbol or name identical to some value, variable, or expression. Different names for the same variable can be used in **ADL. Identities** are declared in the identity list.

Example 3.6.1 - Identities

In this example "a" and "b" are declared identical. "e" and the range of bits 1 through 4 in array d are identical.

(iden (a b) (e (d [1 % 4])))

Identities can be declared for expressions. By using an identity for an expression, the same complicated expressions and fields of bits do not have to be repeatedly typed. Expressions are written in the same format as in the main program. Variables referred to in expressions are declared normally. The identity name itself doesn't need to be declared anywhere else. To avoid error and confusion, an identity name should not be used as variable names or a program name.

Example 3.6.2 - Identities and Expressions

In this example "id1" is declared to be identical to the expression - (or (x [1 % 1]) a):

(iden (id1 (or (x [1 % 1]) a)))

The **DIADES** system replaces identities by their corresponding values or expression during compilation of the **ADL** program. The identity list is similar to #define statements in C.

3.7 SYMB LIST

A single statement can be given a symbolic name in **ADL.** The symbolic name can be used in place of the statement in the program. Use of symbolic names reduces the amount of repetitive typing in some programs. In the **IDEN** list only a single name was used to represented the identity. In the **SYMB** list, a whole expression or statment can be used for the symbol. Symbolic names and their statements are declared in the **SYMB** list.

The structure of the **SYMB** list is similar to the structure of the **IDEN** list. Symbolic names should not be used as variable names or program names. A symbolic name does not have to be a single element. It can be a combination of elements. This means whole statements can be replaced automatically with other statements. Variables in the statement are declared normally.

Example 3.7.1 - Symbolic List

```
(symb (c1 (k := (k + 1)))
      ((a := (c + b)) (adder1 (a c b))) )
```

In the first example, the c1 symbol would be replaced by the statement (k := (k + 1)) in the program. In the second example, the statement (a := (c + b)) is replaced by the subroutine call (adder1 (a c b)). The subroutine could be a structural description of a special adder.

3.8 SUBROUTINES

Subroutines can be used in **ADL.** The subroutine itself is listed at the end of the **ADL** program. Subroutine names and parameters are listed in the declarations

section. There are two types of subroutines, macros and blocks. A subroutine can be structural or behavioral. Macros and blocks can be behavioral. A logmacro subroutine is a special form of the macro and describes structure only. A macro is a sequence of statements which is represented by a single macro call statement in the main program. Macros share the same resources with the main program. A block is a separate system. **DIADES** generates a separate data path and control unit for the block.

Each subroutine is declared using the subroutine definition list. A subroutine is defined by its type, symbol, name, fix flag, and a list of parameters. The symbol, which is the same type of parameter appearing at the beginning of the main declaration block, must be unique to each subroutine and cannot be the same as any other symbols in the program. The fix flag protects the subroutine against any transformations. Transformations optimize the program in various ways. If for some reason, the user doesn't want the subroutine to be modified, the fix flag is enabled.

The parameter list is used to indicate how data is passed between the subroutine and the main program. Constants and fixed numbers can be passed to subroutines and are declared here. **ADL** differs from Pascal-type languages in this respect. The data direction of a variable is specified in the subroutine listing, not here in the main declaration section.

Variable parameter names can be a legal name and correspond to variable names in the subroutine listing. The positions of variables and other parameters must correspond with the positions in the subroutine listing.

Example 3.8.1 - Subroutine Declarations

(subr (macro b example_macro (a b))

(block c example_block (4 a b c)))

The basic grammar for subroutine declarations is:

<subroutine declaration> ::= (subr <subroutine definition>+) ;

<subroutine definitions> ::= (<subroutine type> <symbol>
<name> <fix flag>
<parameter list>) ;

<subroutine type> ::= macro | block | logmacro | logblock ;

<parameter list> ::= (<parameter>+) ;

<parameter> ::= <number> | <vector> | <variable> ;

<fix flag> ::= fix | empty ;

3.9 STATEMENTS AND EXPRESSIONS

3.9.A Introduction

ADL is a procedural language. Each statement is executed sequentially, except in parallel constructs. **DIADES** generates a machine from the program. The machine begins execution with the first statement in the program. Unless there is a loop, the last statement in the program is executed last and the machine stops. Each statement is assumed to execute in one clock cycle and is considered one operation. **DIADES** makes allowances for those statements taking more than one cycle to execute.

Control flow graphs are used in several examples to illustrate program fragments. Each node of a graph corresponds to a one cycle operation. Each edge corresponds to the flow of control through the graph. The following example presents the **ADL** program from the introduction. Following it is the corresponding control-flow

graph.

Example 3.9.A.1 - Program and Flow graph

```
line 1: (((adl a example_circuit ....
line 2:              (input (op1 (p k1 8)) (op2 (p k1 8)) )
line 3:              (intern (temp (p k1 8)))
line 4:              (output (answer (p k1 8)))
line 5:    ((start) a
line 6:         10  (temp := (op1 + op2))
liine 7:       (if (temp = 10) then (temp := (op1 - op2)))
line 8:       (answer := temp)
line 9:       (go 10)
line 10: )))
line 11: end
```

Flow graph of the above example is shown in Figure 4.

Numbers in **ADL** are specified as integers or as a vector of bits. Negative integers are allowed but the form is slightly different than in other high level languages. The maximum size of an integer is limited to the size of the variable being used. Otherwise, there is no theoretical limit to the maximum integer size. Negative integers require a sub-expression using a "-" sign and a positive integer contained in a set of parentheses. For instance, -5 is written as (- 5).

Binary numbers are allowed in **ADL** and are referred to as vectors of bits. There is a special form for the specification of bits. The format is shown below:

(vect (1's and 0's))

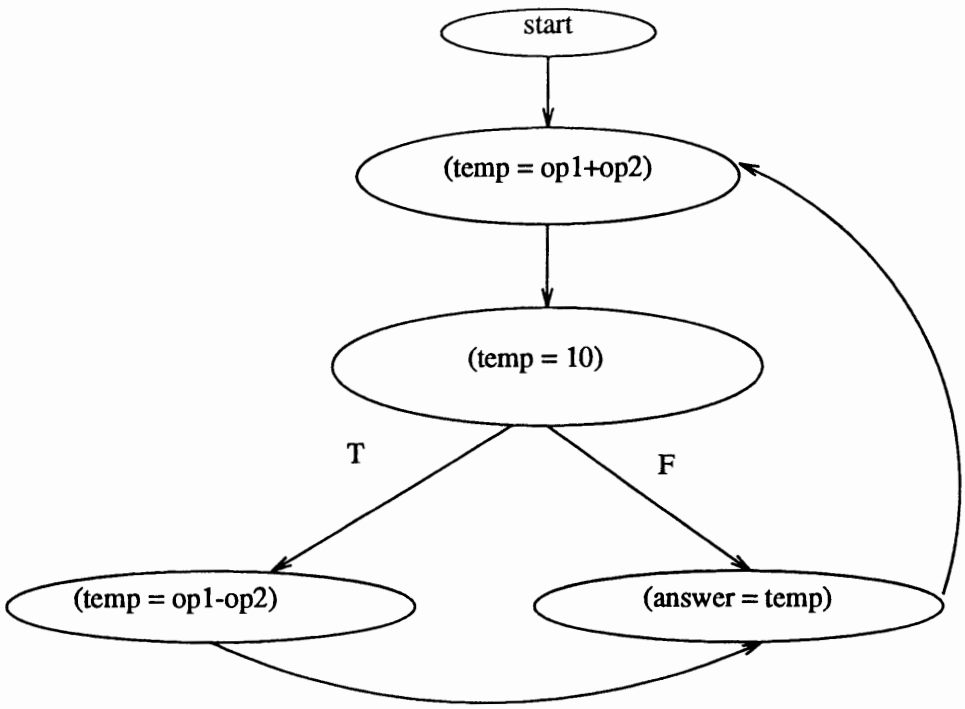

Figure 4. Flow graph of example 3.9.A.1.

The bits are specified as 1's or 0's, each bit separated by a space. The most significant bit is leftmost. Leading 0's are not needed. The maximum size is limited to the size of the variable being used.

There are 3 types of statements; *assignment, conditional-control flow,* and *system control. Assignment statements* include simple variable assignments, arithmetic, and logical statements. *conditional-control flow* statements consist of go, if-then-else, while, and cond statements. *System control* statements control parallelism, process control, and starting and stopping the machine.

The basic grammar for a statement is:

<statement> ::= <label> <statement>

/ <statement> ;

<label> ::= natural number ;

<statement> ::= <system statements> /

<assignment statement> /

<arithmetic statement> /

<logical statement> /

<control flow statement> /

<assertion statement> ;

Each statement is contained within a set of parentheses. Most statements imply their structure but the parentheses are needed to exactly define the limits of statements. Expressions within statements also need parentheses for definition. Each level of expression requires two parentheses. Constants, numbers, and variables do not require parentheses.

Example 3.9.A.1 - Example Statements

(l := (l + 1))

(a := (and l a))

(if (a = 0) then (l := 4) else (l := 1))

(b := (a + (b + c)))

(while (b < 4) do (e := 0)

(b := b - 1))

3.9.B Labels

Statements can have **labels. Labels** are used to mark the destinations of "go" statements. **"go"** statements are similar to "goto" statements in BASIC. A **label** is always a number and letters are not permitted. A **label** is placed on the same line as the statement it refers to but outside the parentheses for that statement. A **label** is inside of any parentheses surrounding a block of statements. Each labelled statements must have a different **label** number.

Example - Statements with Labels

```
10   (a := 5)
20   (if (a < 5) then (a := (a + 1))
                      (go 20)
                 else (a := 100)
                      (go 30)
                      )
30   (c := 5)
```

3.9.C Assignment Statements

The most basic statement is the assignment statement. An assignment means the transfer of the value of an expression to a variable. An expression can be a number, constant, variable, or other operation. The user can split data from one variable into two parts and two variables with an assignment statement. This is referred to as data path splitting. Two variables can be combined into one vector of bits and the new value transferred to another variable. When a program is implemented in hardware, the assignment means the loading of a register with data.

The basic grammar for assignment statements:

<assignment statement> ::= (<left hand side>
<assignment operator>
<expression>) ;

<left hand side> ::= <variable> / <concatenation of variables> ;
<concatenation of variables> ::= (<variable> @ <variable>) ;

<assignment operator> ::= == / := / =: ;
<expression> := <constant> / <variable> / <arithmetic expression>
/ <logical expression> / <predicate>
/ <expression> @ <expression> ;

<predicate> ::= (<expression>
<relation operator>
<expression>) ;

Variables on the right hand side can not be output variables. The output of an output variable goes out of the system and is not available for transfers within the system. However, by also declaring an output variable as an intern variable, data from the output variable can be transferred within the system.

The left and right hand statements do not have to represent vectors of bits with the same size. It is possible to transfer values between variables with different sizes. If the right hand side has more bits than the left hand side, the right hand side is truncated to the correct size by dropping the extra leftmost bits from the connection between the two blocks, (Figure 5.). If the left hand side is bigger than the right hand side, extra 0 bits are added to the right hand side. These extra bits are the "most

significant bits" and are added to the left side. Bits are added until the length of the right hand side matches the length of the left hand side, (Figure 6.).

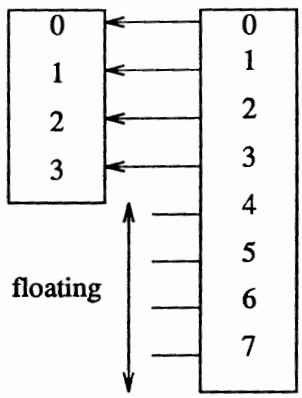

Figure 5. Transfer of values between variables.

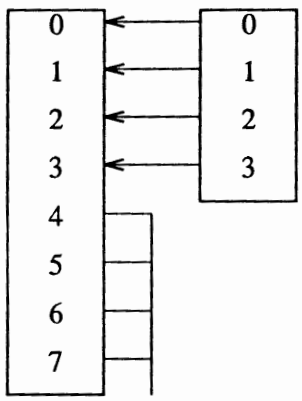

Figure 6. Transfer of values between variables.

Truncation of a variable changes the value of the destination while adding extra bits does not. The user should avoid these problems by using concatenation operations where possible.

Concatenation is a way to combine data moved between variables and expressions. Concatenation in **ADL** has two meanings. One meaning is the combining of two data paths into one data path. The merged data path is input to some hardware

element. The other meaning is the splitting of one data path into two data paths. The two data paths are input to different hardware elements. The format for the two meanings of the concatenation symbol is identical.

The concatenation operator for **ADL** is the '@' symbol. The format for concatenation is as follows:

(<var1> @ <var2>)

or

(<expression> @ <expression>)

<expression> := <constant> / <variable> / (- <variable>)
/ <arithmetic expression>
/ <logical expression> / <relation>
/ <expression> @ <expression>
/ <logical function> ;

The result of the concatenation can be treated as a variable for assignments and expressions. The output from a concatenation is always input to some element or pair of elements. It can either be a lhs(lefthand), or rhs(righthand).

Example 3.9.C.1 - Concatenation

(A := (B @ C))

"A" is an 8 bit variable, B and C are 4 bit variables. The hardware implementation of this operation is to combine the outputs of the B and C registers and transfer the values into the a register, (Figure 7.). B provides the 4 low order bits and C provides the 4 high order bits into A. Example 3.9.C.2. - Data Path Splitting

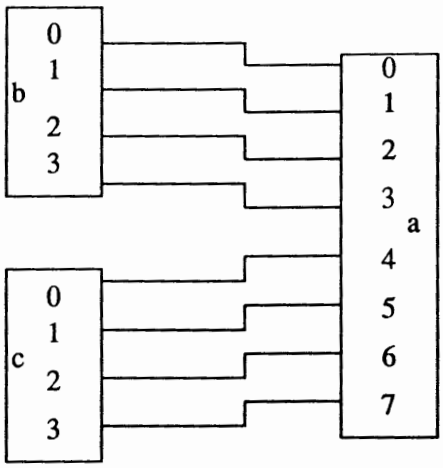

Figure 7. Concatenation operation.

((B @ C) := A)

A, B, and C are variables as above. The hardware representation of this operation is to split the output of register A into the B and C registers, (Figure 8.). B takes the 4 low order bits from A and C takes the 4 high order bits.

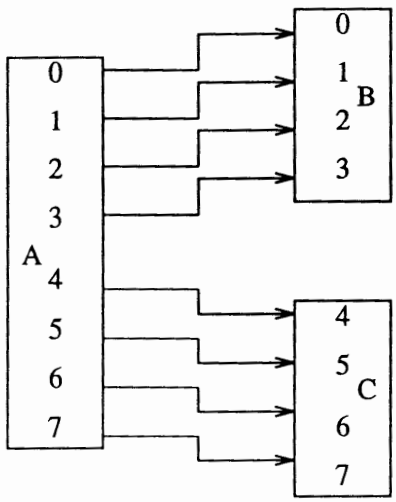

Figure 8. Concatenation operation.

Specific fields of bits can be used with concatenation operations. This allows greater flexibility in controlling the flow of data. If the sizes do not match, then the standard defaults apply. However, the user should be careful to avoid these situations and the system will give a warning.

Example 3.9.C.3.- Concatenation with Bit Fields

((A [3 % 8]) := ((B [2 % 3]) @ (C [0 % 3])))

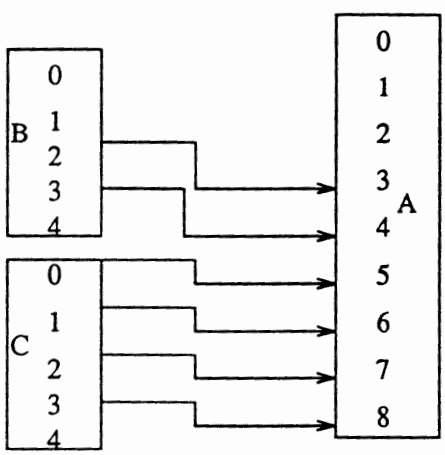

Figure 9. Concatenation with bit fields.

A, B, and C are variables. The hardware representation of this operation is to transfer the specified bits from the B and C registers to the A register, (Figure 9.).

3.9.D Arithmetic Statements

Arithmetic statements in **ADL** are composed of an assignment statement and an arithmetic expression. An expression is composed of operators and operands. Operands can be expressions, variables, constants, and numbers. Arithmetic operations use an infix notation. Multi-level expressions are allowed. The order of

evaluation for an expression is controlled by the parentheses.

Data in **ADL** are always unsigned integers. Two's complement interpretation is up to the user. The high order bit is not reserved for a sign bit. If the result of an operation is negative, the result is in two's complement form. For example, if d is a 4 bit variable with the following assignment:

(d := (3 - 5))

then:

(d = 14) -> true

also with the use of a negation operator:

(d = (- 2)) -> true

The maximum value of an integer depends on the number of bits in the variable. The grammar for arithmetic expressions:

<arithmetic statement> ::= (<left hand side>
<assignment operator>
<arithmetic expression>) ;

<left hand side> ::= <variable> | <concatenation of variables> ;

<concatenation of variables> ::= (<variable> @ <variable>) ;

<assignment operator> ::= == | := | =: ;

<arithmetic expression> ::= (<expression>
<arithmetic operator>
<expression>) ;

<expression> := <constant> | <variable> | (- <variable>)

| <arithmetic expression>

| <logical expression> | <relation>

| <expression> @ <expression> ;

<arithmetic operator> := + | - ;

Example 3.9.D.1 - Arithmetic expressions

(a - c)

(a + (- b))

((a + 1) - (x / y))

*(((- x) + ((a * (- c)) - b))*

(b + 3)

*((5 * 6) + (a - 3))*

In the above example, only arithmetic operators are used. Although the grammar for expressions is extensive, the only other operators allowed in arithmetic statements are logical operators and concatenation.

Example 3.9.D.2 - Complex Arithmetic Expressions

((a + 1) - (and x y (r @ u)))

((or a b c) + (and (x + y) b))

There is no type checking in **DIADES.** Source operands and destination variables can be different sizes, creating some problems. Two 32 bit operands can be added and stored to an 8 bit operand. Only the lower 8 bits of the result will be saved and the upper 24 bits lost. **DIADES** automatically adds an overflow bit to the result of any addition or subtraction operations. Thus, the addition of two 8 bit operands

results in a 9 bit result.

Subtraction operation converts the subtrahend into a two's complement representation and adds it to the minuend. If the result is negative, the result will be a two's complement number. If the subtrahend is the result of another subtraction operation and is negative, it is still converted into a two's complement and added to the minuend.

Multiplication and division operations are included in **ADL** even though they are not implemented in the **DIADES** system. There are no corresponding hardware elements for multiplication and division so these two operations cannot be used.

The size of the result of an arithmetic operation is always equal to the size of the largest operand. Problems can arise when two large numbers are used and the result is too large for the destination.

3.9.E Logical Operations

Logical statements in **ADL** are composed of an assignment statement and a logical expression. A logical expression takes any number of operands and maps them onto a logical operator. Any expression can be an operand. The number of operands is unlimited except for "exor" which takes two operands and "not" which takes one operand. Logical expressions use a prefix notation.

Example 3.9.E.1 - Logical Operation

If all the operands of an expression have one bit, the result of the logical operator is one bit. If the operands have different sizes, the result of the logical operator is equal to the size of the largest operator. Smaller operands have 0s added to the more

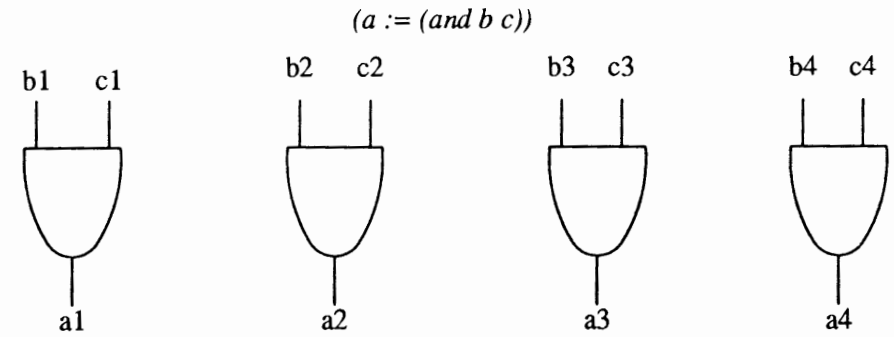

Figure 10. Logical operation.

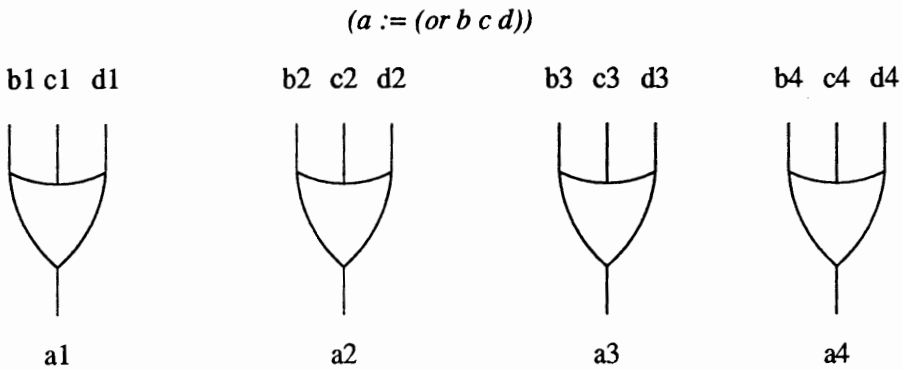

Figure 11. Logical operation.

significant sides.

There are two types of variables in **ADL,** "d" and "p" types. The "d" type is a logical variable and is not used for numbers. It is always one bit. The "p" type is an array of bits and can be used for numbers. The important difference between them is their interpretation in logical expressions. "p" type operands are mapped onto the logical operator. The operator is applied to each set of bits from the operands. "d" type operands have one bit. If the other operands are "p" type, the "d" bit is projected onto the bits of the other operands and the logical operator is applied.

Example 3.9.E.2 - Logical Expressions

In this example:

"p" type:

a = 0111 = 3

b = 0110 = 6

c = 011101 = 29

d = 10 = 2

e = 1 = 1

"d" type:

f = 0

g = 1

(not a) = 1000

(and a b) = 0110

(or c d) = 011111

(and e d) = 00

(and e f) = 0

(nor (a + b) c) = 000010

(and (a + b) (c - d)) = 001001

The grammar for logical statements:

<logical statement> ::= (<left hand side>

<assignment operator>

<arithmetic expression>) ;

<left hand side> ::= <variable> | <concatenation of variables> ;

<concatenation of variables> ::= (<variable> @ <variable>) ;

<assignment operator> ::= == / := / =: ;

<logical expression> ::= (<logical operator>
<expression>+) ;

<expression> := <constant> / <variable> / (- <variable>)
/ <arithmetic expression>
/ <logical expression> / <relation>
/ <expression> @ <expression> ;
<logical operator> := and / nand / or / nor / exor / not ;

3.10 CONTROL FLOW STATEMENTS

Control flow statements are used to control execution of branches in the program. **Go** statements branch unconditionally.

If-then-else and **while-do** statements allow the program to have several branches. The branch executed depends on the value of the predicate in the control statement. **Cond** statements select one of several branches depending on the value of a predicate at the beginning of each branch. **Wait** statements hold the system in a loop for a specific time or until some input value changes.

The basic grammar of control flow statements is:

<control flow statement> ::= <go statement> /
<if then else statement> /
<while statement> /
<cond statement> /

<wait statement> ;

Each statement, except go, uses a predicate to control branching. Predicates are expressions which have two values, true and false. True is represented by one bit with value 1 and false by one bit with value 0. A predicate consists of one or more relations linked with a logical connective such as "and" or "or". A relation consists of a relational operator and two operands. An operand can be a number, constant, variable, or expression. The basic grammar for a predicate is:

<predicate> ::= (<expression>
<relational operator>
<expression>)
or
(<expression>) ;

<expression> := <constant> / <variable> / (- <variable>)
/ <arithmetic expression>
/ <logical expression> / <predicate>
/ <expression> @ <expression> ;

<relational operator> := = / < / > / ;

Example 3.10.1 - Predicates

a = 3 b = 1 c = 2

(a = 3) -> *true*

(a = b) -> *false*

(a > 7) -> *false*

(b < 8) *-> true*

(a = (b + c)) *-> true*

((5 - a) > (c + 4)) *-> false*

(and (a = 3) (b < 8)) *-> true*

(or (a = b) (a > 7)) *-> false*

(not (b < 8)) *-> false*

(nor (a = (b + c)) (b = 1)) *-> false*

3.10.A Go Statements

Go statements are used to unconditionally branch to a labelled statement. **Go** statements and their corresponding destination statements can occur anywhere in the program. Jumps out of **if-then else** statements and while loops are allowed. Although it is not a good programming practice, the destination statement can be inside of a while loop. The "value" of a **label** cannot be created using an expression. For example: (i := 99) and (go i) is invalid. The basic grammar for the go statement is:

<go statement> ::= (go <label>) ;

Example 3.10.A.1 - Program with Go Statements

```
(((adl a example
      (input  (a (p k1 8)) (b (p k1 8)) (c (p k1 8)))
      (intern (j (p k1 8)) (k (p k1 8)) (l (p k1 8)))
      (output (x (p k1 8)) (y (p k1 8)) (z (p k1 8)))
 )
```

```
((start) a
10    (j := (a + 1))
      (go 30)
20    (k := (a + b))
      (go 40)
30    (l := (b + c))
      (go 20)
40    (x := (j - k))
      (go 10)
)))
end
```

The flow graph of the above program is shown in Figure 12.

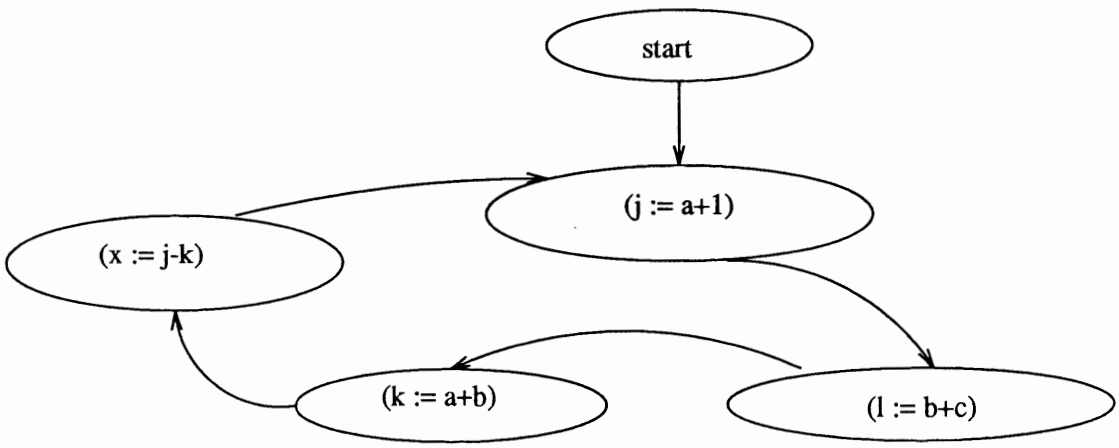

Figure 12. Flow graph for example 3.10.A.1.

3.10.B If-Then-Else Statements

There are two forms of the conditional **if** statement: with an "else" and without an "else". The statement consists of an **"if"** keyword, followed by a predicate. The predicate is followed by a "then" keyword. If the predicate is true, the statements following it are executed. The "else" keyword follows those statements. If the predicate is false, statements following the "else" keyword are executed. If there is no "else", statements after the if-then block are executed. The basic grammar of the **if** statement is:

<if statement> ::= (if <predicate> then <statement>+) |

or

(if <predicate> then <statement>+else <statement>+) ;

Unlike C or Pascal, there are no explicit symbols defining the extent of the **if-then-else** statement. In Pascal, begin and end keywords mark the sequence of statements being executed. In **ADL** the "then" keyword and either the "else" keyword or right parenthesis mark the sequence of statements executed if the predicate is true. The "else" keyword and the right parenthesis mark the sequence of statements executed if the predicate is false.

Example 3.10.B.1. - Program with If Statements

```
(((adl a example
      (input  (a (p k1 8)) (b (p k1 8)))
      (intern (j (p k1 8)) (k (p k1 8)))
      (output (x (p k1 8)) (y (p k1 8)))
 )
((start) a
```

```
(if (a = 3)
  then
    (j := (a + b))
    (k := (and j (or (a + b) a)))
  else
    (x := 5)
)
(y := 8)
(if (k > (j + a))
  then
    (y := 32)
)
)))
end
```

The flow graph of the above program is shown in Figure 13.

3.10.C While Loops

ADL supports **while** loops. A while statement consists of the **while** keyword followed by a predicate and the **do** keyword. Statements between the **do** keyword and the right parenthesis are executed repeatedly until the predicate is false. If the predicate is false before the start of the loop, the statements in the loop are not executed. The basic grammar is shown below:

<while statement> ::= (while <predicate> do <statements>+) ;

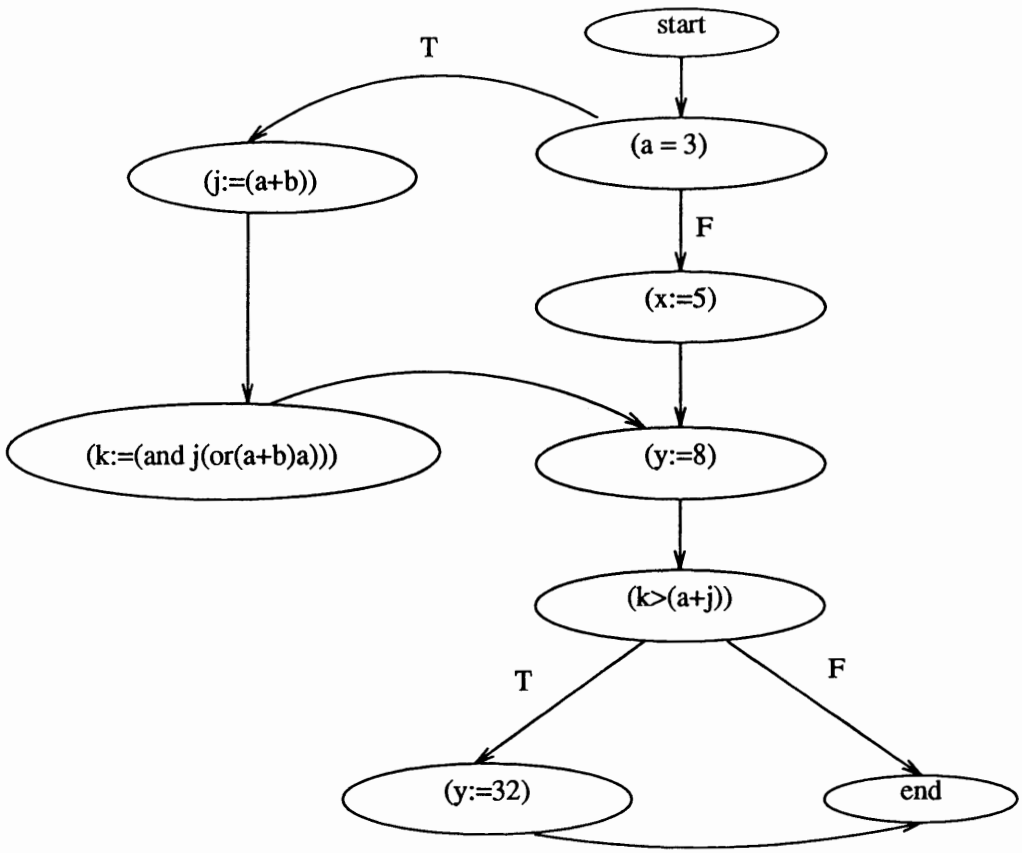

Figure 13. Flow graph for example 3.10.B.1.

Example 3.10.C.1 - Program with While Loop

```
(((adl a example

      (input  (a (p k1 8)) (b (p k1 8)))
      (intern (j (p k1 8)) (k (p k1 8)))
      (output (x (p k1 8)) (y (p k1 8)))
 )
((start) a
```

```
( while (a = 3) do
    (j := (a + b))
    (k := (and j (or (a + b) a)))
)
(y := 8)
(while (k > (j + a)) do
    (y := 32)
)
)))
end
```

The flow graph of the above program is shown in Figure 14.

Any statements can be executed in the while loop except **return, drop,** and **stopadl.** Jumps into and out of the loop with the **go** statement are allowed. Nested **while** loops are allowed.

If the predicate variables are input variables, they do not have to be initialized. If they are **intern** variables, they should be initialized before the **while** statement is executed. The predicate is evaluated first. If true, the statements within the loop are executed and the predicate is evaluated again. If the predicate is false, execution continues with the statement following the **while** statement.

Normally, the predicate variable should change in the body of the loop so that the loop is exited. This is true for **intern** variables. It is not true for **input** variables because their values are changed from outside the system and the user cannot change those from within the program.

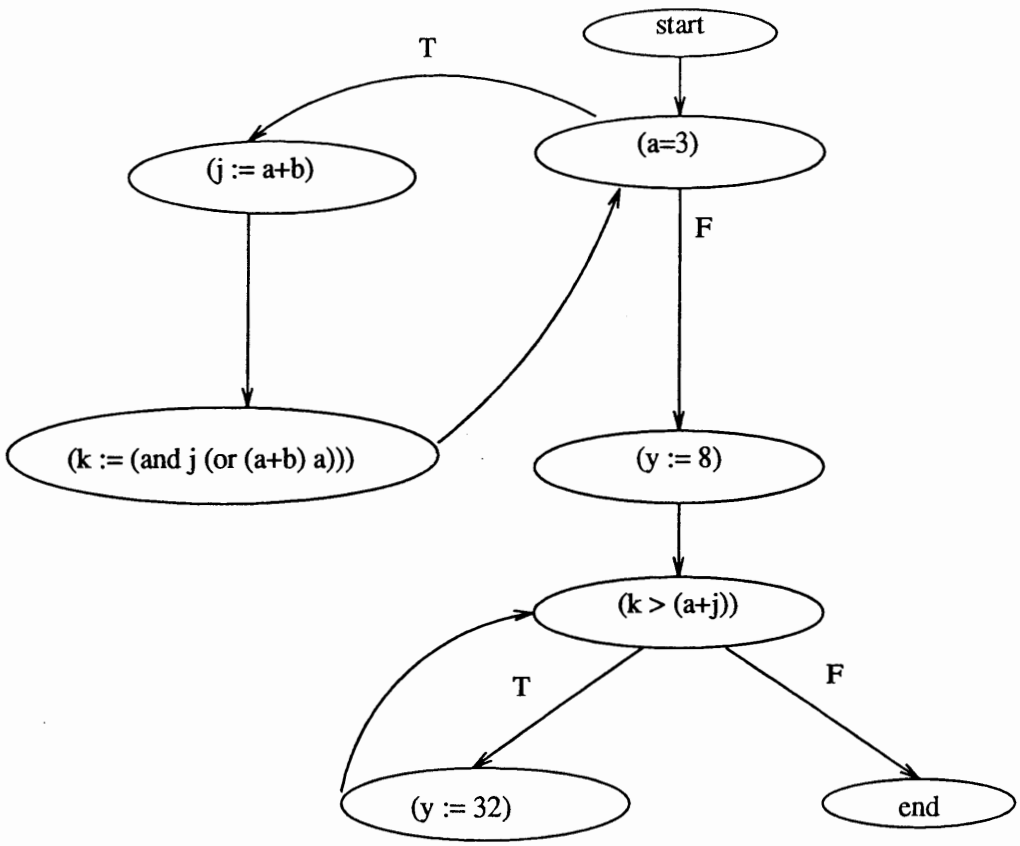

Figure 14.Flow graph for example 3.10.C.1.

Example 3.10.C.2 - Nested While Loops

```
(((adl a example
      (input  (a (p k1 8)) (b (p k1 8)))
      (intern (j (p k1 8)) (k (p k1 8)))
      (output (x (p k1 8)) (y (p k1 8)))
 )
((start) a
 ( while (a = 3) do
   (j := (a + b))
```

```
(k := (and j (or (a + b) a)))
    (while (j > 10) do
        (j := (j + 1))
        (x := (nor a b c))
    )
)
)))
end
```

The flow graph of the above program is shown in Figure 15.

3.10.D Cond Statement

The **cond** statement is a an extension of the **if-then-else** statement. Within the **cond** statement can be several branches, but only one is executed. Each branch has its own predicate controlling execution of that branch. When a **cond** statement is encountered, the predicate of the first branch is executed. If it is true then that branch is executed and program execution continues with the next statement after the **cond** statement. If the first predicate is false then the next branch is executed. This process goes on until one of the branches is executed or the end of the **cond** statement is reached. There is no "default" branch. There must be at least two branches in a cond statement. Jumps out of a **cond** statement using the **"go"** statement are allowed. The basic grammar for the **cond** statement is presented below:

<cond statement> ::= (cond <cond branch>+) ;

<cond branch> ::= (<predicate> <statement>+) ;

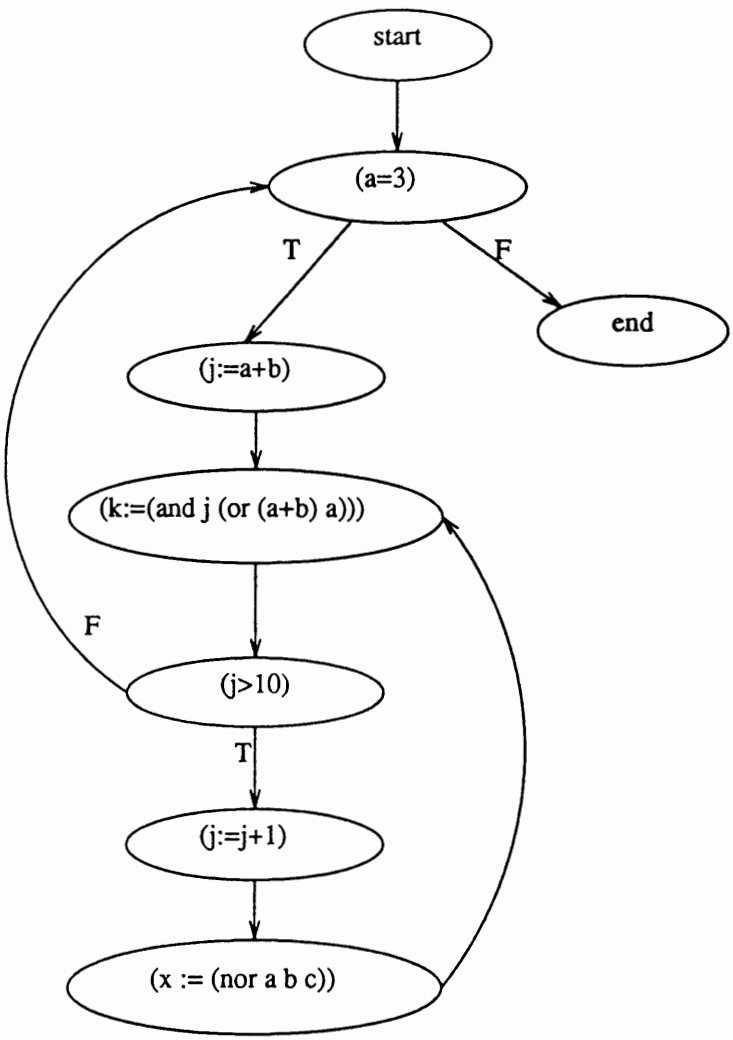

Figure 15. Flow graph for example 3.10.C.2.

Example 3.10.D.1 - Cond Statements

(((adl a example

(input (a (p k1 8)) (b (p k1 8)) (c (p k1 8)))

(intern (j (p k1 8)) (k (p k1 8)) (l (p k1 8)))

(output (x (p k1 8)) (y (p k1 8)) (z (p k1 8)))

```
((start) a
10    (cond ((b = 0)  (x := 1) )
            ((not (b = 0)) (y := 0) (z := 0) ) )
      (cond ((a = 0) (go 10) )
            ((a = 1) (while (j > 10) do (j := (j + 1))
                                        (x := (nor a b c))
                  ) )
            ((c = 23) (l := 1) )
            ((j = 3) (x := 0) (k := (k + 2) )
      )
)))
end
```

The flow graph of the above program is shown in Figure 16.

3.11 SPECIAL STATEMENTS

This section describes statements which are unique to ADL. They provide more control over hardware operations. These statements include a **wait** statement which holds the digital system in a timing loop until some condition is met and **set** and **reset** statements which assert output values.

3.11.A A Wait Statements

The **wait** statement is used to hold the digital system in a timing loop for a fixed amount of cycles or until some external condition is met. The **wait** statement with a

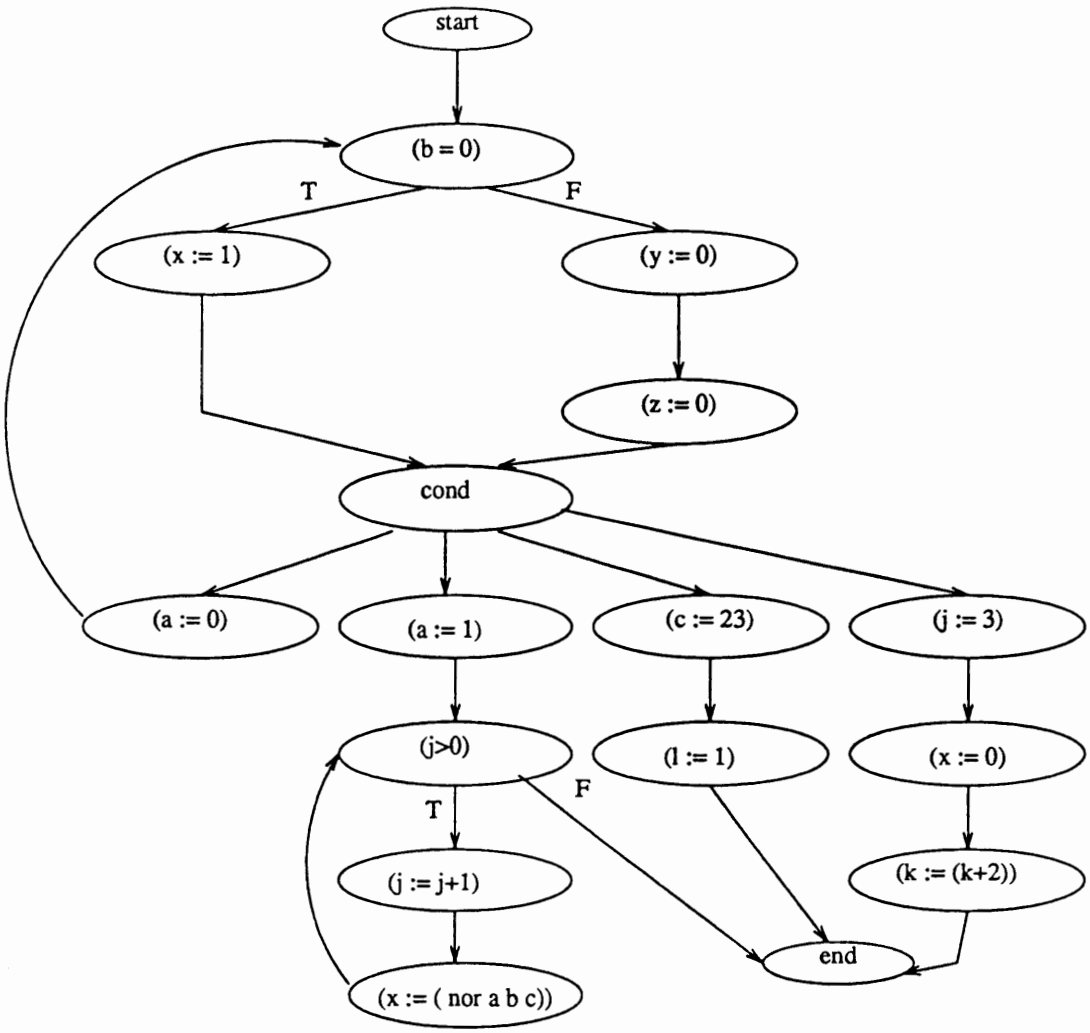

Figure 16. Flow graph for example 3.10.D.1.

number is used to hold the system for a fixed time. The **wait while** statement with a predicate is used to hold the system until an external condition is met. Variables used in a **wait while** statement must be **input** variables because internal variables will not change during the timing loop. The **wait** statement is an extension of a **for** loop or **while** loop. The basic grammar for this statement is presented below:

<wait statement> ::= (wait <number>) |

(wait while <predicate>) ;

Example 3.11.A.1 - Wait Statements

(wait while reservoir_full)

(wait while (and (x [1 % 1]) r))

(wait 1000)

3.11.B Set and Reset Statements

Set and **reset** statements control the turning on and off of many one bit signals. **Set** means an output variable is 1 and **reset** means an output variable is 0. This application is common in industrial control circuits. The basic grammar is shown below:

<set and reset statements> ::= (set <logical variable>+) |

(reset <logical variable>+) ;

Only logical variables can be **set** and **reset** because they are one bit signals. Once a variable is **set** or **reset** , it holds that value until the variable is changed.

Example 3.11.B.1 - Set and Reset

(set a b c)

(reset x y z)

3.11.C System Control Statements

System control statements are divided into two groups, statements controlling the starting and stopping of the digital system, and statements controlling parallel

operations.

3.11.D Parallel Execution Statements

In **DIADES** the user can design a system with parallel operations. Parallel operations use the same control unit but multiple hardware elements are enabled at the same time. An **ADL** program can be written so that individual statements or blocks of statements operate in parallel. Multiple statements can be directed to execute in the same cycle. An algorithm can be split into parallel blocks of statements. Control statements join parallel blocks back into one process.

3.11.E A Multiple Statement Execution

The **sim** instruction is used to execute more than one assignment instruction in the same cycle. **Sim** stands for simultaneous execution of statements. All statements within the scope of the sim instruction are executed in the same cycle. Only assignment statements can be executed. The basic grammar for the **sim** instruction is:

<sim instruction> ::= (sim <assignment instruction>) ;

Example 3.11.E.1 - Sim Instructions

In this example, x and k are incremented in parallel.

```
(sim (x := (x + 1))
     (k := (k - 1))
)
```

In this example, x is loaded with 1, a is loaded with 5, and b is anded with x and a.

```
(sim (x := 1)
        (a := 1)
        (b := (and x a))
)
```

At least two statements must be within the **sim** instruction. If the same variable appears on the left side of one assignment statement and on the right side of another in the same **sim** instruction, the **TAG** compiler will print a warning about conflicting variables. This is not an error but a warning to possible problems with this kind of instruction in the next phases of system design. An warning is also printed if the number of bits doesn't match in a field specification.

3.12 PARALLEL PROGRAM EXECUTION

An algorithm is a process. Using parallel control instructions, a process can be split into two or more branches. Each branch executes in parallel with the other branches. Each branch operates independently of the other branches and communication between branches is through internal variables. When one or all branches are finished, they can be joined together into one process or just stop. A branch can stop when it is finished, wait for other branches to finish, or cause other branches to stop. Fork is the name of the branching instruction and the basic grammar is described below [10].

<fork instruction> ::= (fork <branch>+ <branch control>) ;

<branch> ::= (<instructions> <drop control>) ;
<drop control> ::= (drop) ;

<branch control> ::= dand | dexor ;

The **fork** instruction is composed of the keyword **fork,** followed by each branch which is enclosed in a set of parentheses. After the branches are described, the branch control parameters are described. Control parameters indicate how control passes from the branches to a single process.

There must be at least two branches in a **fork** instruction. Each branch is made up of almost all **ADL** statements, including parallel control statements. The restrictions are the following:

1. **Labels** can not be placed on the last statement in a branch.
2. Jumps out of, into, and within the branch are allowed. Jumps out of and into the fork statement are also allowed. However, programming like this causes problems and the user should be very careful.
3. Instructions **drop, return,** and **stopadl** cannot be the only instruction in a branch.

Branches can be joined into a single process. When one or more branches have completed execution, the overall execution process continues with the next statement after the **fork** statement. The branch control instructions indicate how this is done. The **drop** instruction individually controls each branch and appears as the last statement in a branch. **Dand** and **dexor** instructions control all branches and are normally placed in the **fork** statement after all the branch statements.

1. **drop** - When the branch is finished executing and drop instruction is reached, control is terminated in this branch. Program control is left to the other branches. If all the other branches have **drop** instructions, then

the execution of the next statement after the **fork** takes place when the last branch has finished. If the other branches are controlled by **dand** or **dexor** instructions and finish before the branch with the **drop,** the branch with the **drop** keeps executing in parallel with the rest of the program.

2. **dand** - When any of the branches is finished executing, this instruction is reached and execution of the next statement after the **fork** does not take place until all branches have completed execution. If a branch has a **drop** instruction, that branch is ignored for the scope of the **dand.**

3. **Dexor** - When any of the branches is finished executing, this instruction is reached and control passes to the next statement after the **fork.** All other branches stop execution immediately except for branches with **drop** instructions. These will continue executing until finished. There are some special cases where parallel branches continue executing after the **dexor** instruction is reached.

Example 3.12.1 - Fork Example

```
(((adl a example382
      (input  (a (p k1 8)) (b (p k1 8)) (c (p k1 8)))
      (intern (j (p k1 8)) (k (p k1 8)) (l (p k1 8)))
      (output (x (p k1 8)) (y (p k1 8)) (z (p k1 8)))
 )
((start) a
111   (x := 10)
      (fork
```

```
        ((x := 4) (go 111))
        ((y := 33) (x := (b + 34)) (drop))
        ((z := (and a b)) (k := (j + e))
        ((j := 45) (while (j > 23) do (j := (j - 1))))
        dexor
    )
)))
end
```

The flowgraph of the above program is shown in Figure 17. The first branch ends with the jump out of the branch and **fork** statements to some other point of the program. The second branch ends when the two assignments are completed and the **drop** instruction is reached. The third and fourth branches are controlled by the **dexor** instruction. When one of them finishes, the other branch is stopped and the program execution continues with the first statement after the fork instruction. In this particular example, it is obvious that the third branch finishes first. The variable x will have the last value calculated before the branch was stopped.

When the **dexor** instruction is reached within a **fork** statement, all other branches within the scope of the **fork** statement stop. The **dexor** instruction only affects statements inside the **fork** statement. If a jump is made out of the **fork** statement to another section of the program, that section will be executing in parallel with the branches inside of the **fork** statement. When one branch is finished and the **dexor** instruction is reached, branches within the **fork** statement stop but the section outside of the **fork** statement continues to execute.

There is a special form of the **dexor** instruction. If labels are placed at the

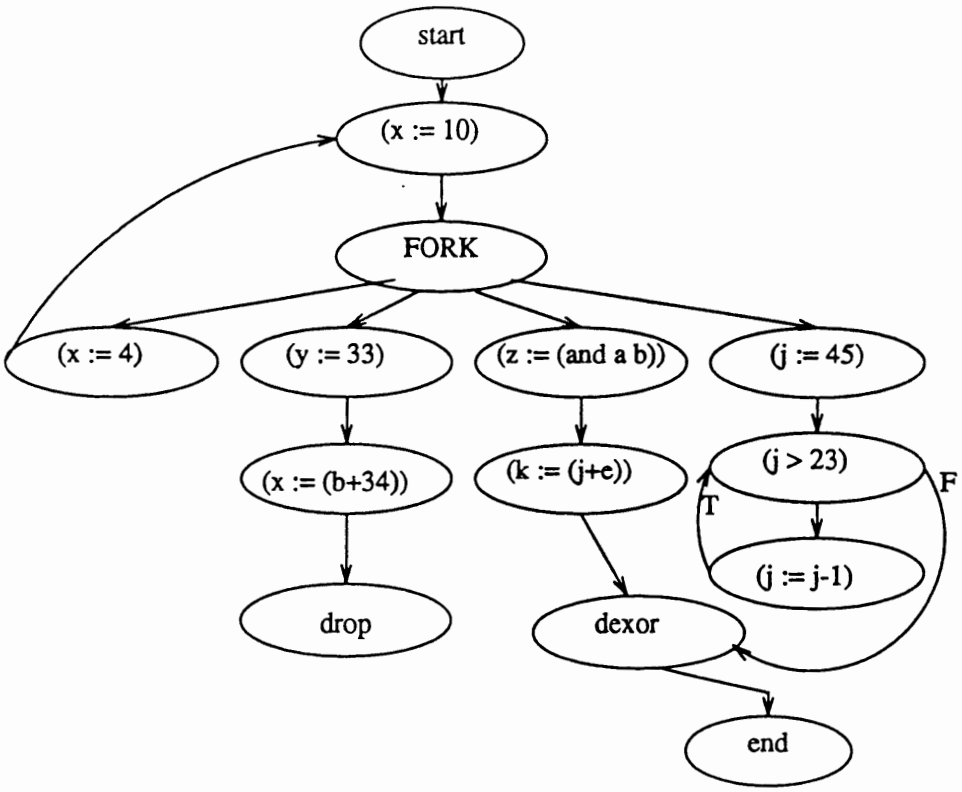

Figure 17. Flow graph for example 3.12.1.

beginning of branches, (within the parentheses), the dexor instruction only affects the control of specified branches. The **dexor** statement is placed outside of the **fork** statement as a separate labeled statement. The syntax for this is shown below:

<label> (dexor <label>+)

Example 3.12.2 - Fork with Dexor Specification

```
(((adl a example
 (input  (a (p k1 8)) (b (p k1 8)) (c (p k1 8)))
 (intern (j (p k1 8)) (k (p k1 8)) (l (p k1 8)))
```

```
        (output (x (p k1 8)) (y (p k1 8)) (z (p k1 8)))
  )
((start) a
1       (x := 10)
        (fork
                ((x := 4) (y := 5) (drop))
                (4 (y := 33) 5 (x := (b + 34)) (go 3))
                (6 (z := (and a b)) (go 3))
                ((j := 45) (go 1)))
3       (dexor 4 5 6 3)
)))
end
```

Tokens in nodes 4, 5, 6, and 3 are canceled if any one of them finishes and a jump is made to the **dexor** statement. All other branches started from this **fork** statement continue. The flowgraph for this example is shown in Figure 18.

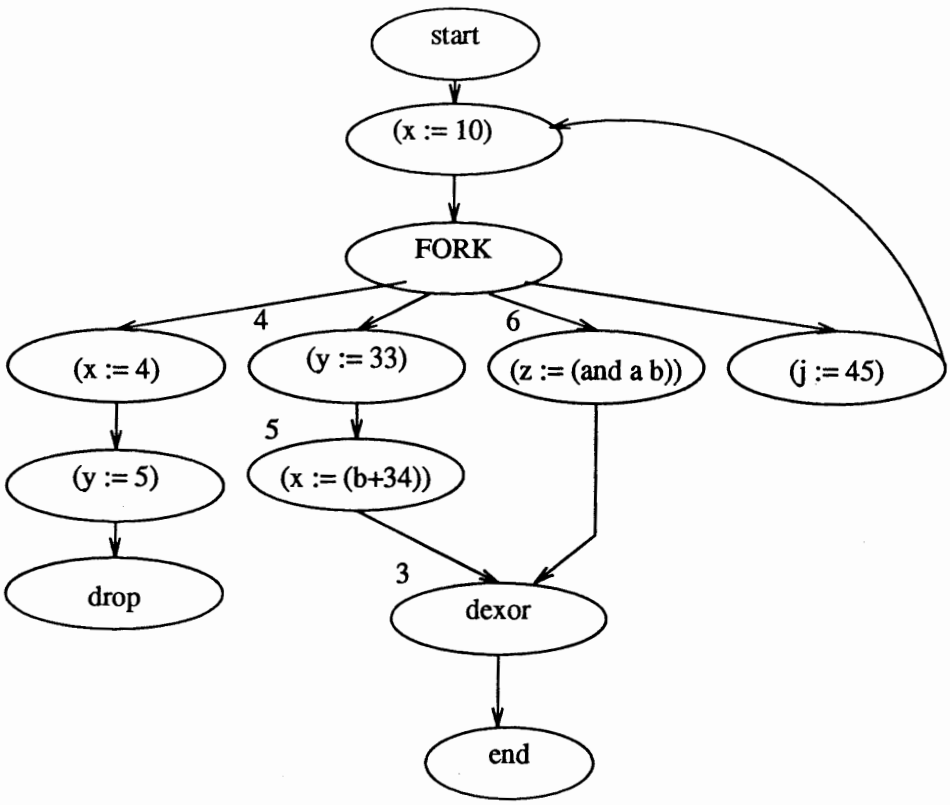

Figure 18. Flow graph for example 3.12.2.

CHAPTER IV

GRAPH LANGUAGE

4.1 INTRODUCTION

GRAPH language is a format used in **DIADES** to represent the behavior of a digital system. The behavior is represented by a specific sequence of program statements. **GRAPH** is based on a standard directed graph. It basically consists of a set of nodes and arrows. A node represents an operation or program statement. An operation can be a variable transfer, arithmetic operation, logical operation, or comparison. An arrow represents the flow of control from one node to another. For example, an arrow from node 1 to node 2 means node 1 is executed first and then node 2 is executed. Control of the digital system is transferred from node 1 to node 2. The **GRAPH** is not evaluated as a whole but a path can be traced through the **GRAPH** representing a sequence of operations.

Control can flow through the **GRAPH** in several ways. Each node is executed sequentially, following the path of the arrows. Some nodes are condition checks. The value of a variable is compared to some expression. The control flow can split into two branches. One branch is when the result of comparison is true and another is when result of the comparison is false. The execution of the branches is mutually exclusive. Only one branch will be executed at a time. Some nodes represent multi-way branches. Each condition is checked until a true one is found. The corresponding branch is executed and the others skipped.

Normally only one node is executed at a time. There is a way to execute nodes simultaneously. Multiple branches can be defined and executed concurrently. The control flow splits to go in parallel through the branches. The parallel control flows can be joined together in different ways resulting in a single control flow.

The data structure for **GRAPH** is more complicated than a simple di-graph. There are several node and arrow types, each with different properties. Each property affects how the **GRAPH** description is interpreted and executed. For example, some nodes are data transfer type operations and some are condition checks. Some node properties affect how arrows are interpreted. Some arrows represent sequential execution, while others represent concurrent execution.

The implementation of **GRAPH** consists of several Lisp lists. The three main lists are the **COPLISSET,** the **NALISSET,** and the **PLISSET.** The **COPLISSET** represents arrows. The **NALISSET** represents nodes containing regular operations and data transfers. The **PLISSET** represents nodes containing comparison operations. Because it is not a pure graph implemented with pointers, all nodes are assigned numbers. Each of the lists uses these numbers to refer to nodes. Some nodes are not listed in either the **NALISSET** or **PLISSET** lists. These nodes usually are used to control execution and are described in the auxiliary lists, **NOLISSET** and **ANLISSET.** Nodes which are identical to some other node are not listed in the **NALISSET** list. Instead, they are listed as being equivalent to other nodes in the **NOLISSET** list. The other lists contain information about variables, the number of blocks, and any structural descriptions. The entire **ADL** description is also contained in one list for reference by some **DIADES** programs that use **GRAPH** language.

4.2 COPLISSET LIST

The **COPLISSET** list contains all the arrows in the **GRAPH** The arrows indicate how control flows through the **GRAPH** in **GRAPH.** Control can flow from one node to another or it can depend on the results of a comparison, (a condition check). An element consists of a symbol indicating the control transfer type, the source node number and the destination node number. These numbers reference to elements of other lists. The formal definition of a **COPLISSET** entry follows:

<COPLISSET element> ::=

(<control transfer type>

<source>

<destination>

) ;

<control transfer type> ::= e | x

| <node number>

| (not <node number>) ;

<source> ::= <node number> ;

<destination> ::= <node number> ;

The source and destination nodes can not be the same. The implementation of a conditional loop requires the use of a dummy node. Dummy nodes are described in the **NOLISSET** list. An x type transfer means control is transferred unconditionally from the source node to the destination node. An e type transfer means there are parallel execution paths from the source node. There will be at least one other arrow with an e transfer type from the same source node to a different destination node. If a node number is given as the transfer type, this means that node is a comparison

operation. The destination node depends on the comparison result. If the result of the comparison is true, then the control transfers to the destination node. If it is false, then the control is not transferred. If the not keyword precedes the node number and is in a sublist, then the control is transferred to the destination node if the comparison result is false. Regular condition checks require two arrows, one for true and one for false.

Example 4.2.1 - Conditional and Unconditional Control Transfer

```
(x 1 2)
(x 2 3)
(3 3 4)
((not 3) 3 5)
(x 4 5)
(x 5 6)
```

The flow graph of the above example is shown in Figure 19.

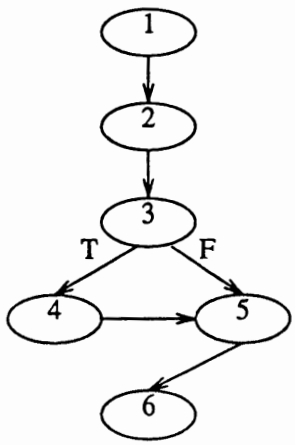

Figure 19.Flow graph for example 4.2.1.

Example 4.2.2 - Parallel Control Transfer

(x 1 2)

(x 2 3)

(e 3 4)

(e 3 5)

(x 5 6)

(x 6 7)

(e 3 8)

(x 8 9)

The flow graph of the above example is shown in Figure 20.

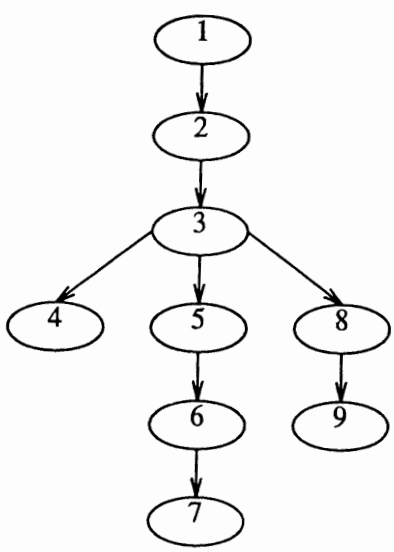

Figure 20. Flow graph for example 4.2.2.

A **cond** structure represents multi-way branching. The **cond** implementation is a little complicated. One comparison node is used to represent the source node for all the comparison operations for the **cond.** The control flow goes through this node and then to the resulting branch nodes. All of the comparison operations in the **cond** are

not listed explicitly in the **GRAPH.** The first comparison node is listed in the **GRAPH.** The control flows through the operation preceeding the **cond** to this node. The first comparison is made. If it is true then control is passed to its corresponding branch. If it is false, then the next comparison is evaluated. There is an arrow pointing to the first comparison node. There are no arrows pointing to the other comparison nodes. There are arrows using the other comparisons between the original source node and the corresponding branch of each comparison. The only way to find these other arrows is to look at all the arrows in a **GRAPH** description for an element with the same source node. In this case that will be the original source node for the comparison. The order of execution for the other comparisons in the **cond** structure is determined by their node numbers.

Example 4.2.3 - Cond Structure

(6 6 7)

(8 6 9)

(10 6 11)

(12 6 13)

The flow graph of the above example is shown in Figure 21.

Nodes 6, 8, 10, and 12 are comparison nodes. The comparison check at node 6 is the source node when transferring control to any of the branches. First the comparison at node 6 is checked. If it is true, then control is transferred to node 7. If it is false, then the control is passed to the comparison at node 8. The only way to know that 8 is the next node is to look at all the arrows in a graph description for an element with the same source node. In this case that is node 6. The order of execution for the

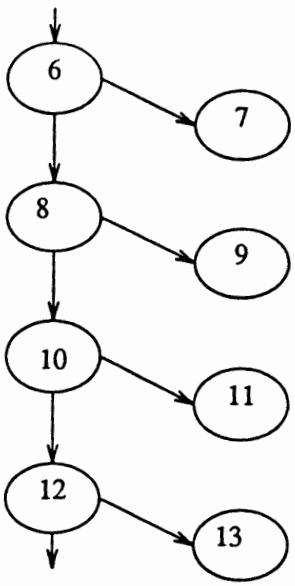

Figure 21. Flow graph for example 4.2.3.

other comparisons in the cond structure is determined by their node numbers.

4.3 NALISSET LIST

The **NALISSET** list describes all of the assignment type operations. Comparison operations are listed in the **PLISSET** list. Each operation consists of an assignment type, a destination for the results of the operation, and an expression. An expression can either be a variable, arithmetic or logical expression, or a concatenation of expressions. An element of the **nalisimp** list has the following form:

NALISSET element ::=

(<node number> (<assignment operator>

<destination>

<expression>

)

) ;

<assignment operator> ::=

:= | =: | == ;

<destination> ::=

<variable> | <concatenation of variables> ;

<concatenation of variables> ::= (<variable> @ <variable>) ;

<variable> ::= <name> |

(<name> [<index>]) |

(<name> [<low bit> % <high bit>])

<expression> := <operand> | <expression>

| <expression> @ <expression> ;

<operand> ::= <constant> | <variable>

| <concatenation of variables> ;

<expression> ::=

(<operator> <operand>+) ;

<operator> ::= plus | minus | times | and | nand

| or | nor | not | exor ;

Example 4.3.1 - Nalisset List

(2 (:= x 10))

Which means (x := 10). Where 2 is the node number.

(6 (:= x (plus b 34)))

Which means (x := (b + 34)). Where 6 is the node number.

(8 (:= z (and a b)))

Which means (z := (and a b)). where 8 is the node numer.

(12 (:= j (plus j (minus 4))))

Which means (j := (j - 4)). Where 12 is the node number.

4.4 PLISSET LIST

The **PLISSET** list contains the node descriptions for comparison operations. A comparison operation, also called a condition check, takes two operands. The result of the comparison is either true or false. This is represented by a single bit, which is sent to the control unit. The format of a PLISSET element is as follows:

<predicate> ::= (<expression>

<relation operator>

<expression>) ;

Example 4.4.1 Plisset List

(3 (and (equal a 5) (equal b 3)))

Which means (if (and (a = 5) (b = 3). The node number for this example is 3.

4.5 NOLISSET LIST

The **NOLISSET** list describes properties of each node. Nodes not described in the **NALISSET** and **PLISSET** can be found here. The **NOLISSET** also indicates

equivalent nodes. Equivalent nodes are defined as two or more nodes which accomplish the same function with the same operands and variables. Equivalent nodes may be executed at different times so the numerical results of the function may be different. For example, nodes 11 and 13 represent the operation, r3 := x. The **NALISSET** will contain the element, (11 (:= r3 x)). There is no element for node 13. Instead 13 is described as being equivalent to node 11. The format of a **NOLISSET** element is as follows:

<NOLISSET element> ::=

(<property> <node number>) ;

<property> ::= contin | cond | start | <node number> ;

There are 4 basic properties. Each node can only have one property. The first item in the NOLISSET is the property. The second item is a node number. This is the node which has the specified property.

Contin indicates a dummy node. The source and destination nodes in an arrow can not be the same. **Wait** operations continuously check a condition. A **contin** node serves as a destination node for the condition check. Another arrow connects the **contin** node with the condition check. A **contin** node doesn't take any time to execute. For example, (contin 16 nil) means that node 16 is a dummy node.

cond indicates a comparison node. Each node that performs a comparison is listed in the **NOLISSET** except for comparisons in a **cond** structure. Only the first comparison of a **cond** structure is listed. The others are not. For example, (cond 6 nil) means node 6 is a comparison. If node 6 was the first comparison in a **cond** structure, then the other comparisons would not be listed.

start indicates a start node. A digital system begins executing with the **start** node. It doesn't take any time to execute. It is defined to make setting up a state machine easier. For example, (start 1 nil) means that the state machine starts with node 1. Node 1 will always be a start node and there will be only one **start** node for each **GRAPH.**

If the property is a node number, then that node is an assignment type node. These nodes are described in the **NALISSET.** The second item in the **NOLISSET** element is another node number. In this case the first and second items in the **NOLISSET** element are the same. For example, (2 2 nil) means node 2 is an assignment type node described in the **NALISSET.**

If a different node number is the second item, then that node is equivalent to the node represented by the first node number. If a given node is listed as being equivalent to some node, then the given node will not be listed in the **NOLISSET.** For example, (11 13 nil) and (11 11 nil) are **NOLISSET** elements. Both are assignment type nodes and node 13 is equivalent to node 11. There is no other element in the **noliset** for node 13. Node 11 has an entry and node 13 does not because 11 comes before 13.

A complete example consisting of an **ADL** program and its **GRAPH** format is shown below.

Example 4.5.1

ADL program

listing

adl

graph

```
subgraph
(((adl c classification
  ((clock (1000)))
  (input ((x [ j ])(p k1 8)))
  (intern (j (p k1 6))(y (p k1 2)))
  (output (y (p k1 2)) (j (p k1 6))))
((start)  c
  (j := 49)
  (while (j > 0) do
    (if (x < 125) then
      (y := 3)
      (go 11))
    (if (x  < 126) then
      (y := 2)
      (go 11))
    (if (x  < 127) then
      (y := 1)
      (go 11))
    (if (x < 128) then
      (y := 2)
    else
      (y := 3))
    11
    (j := (j - 1)))
  (y := 0)
```

(stopadl))))

end

The flow diagram for the above program is shown in Figure 22.

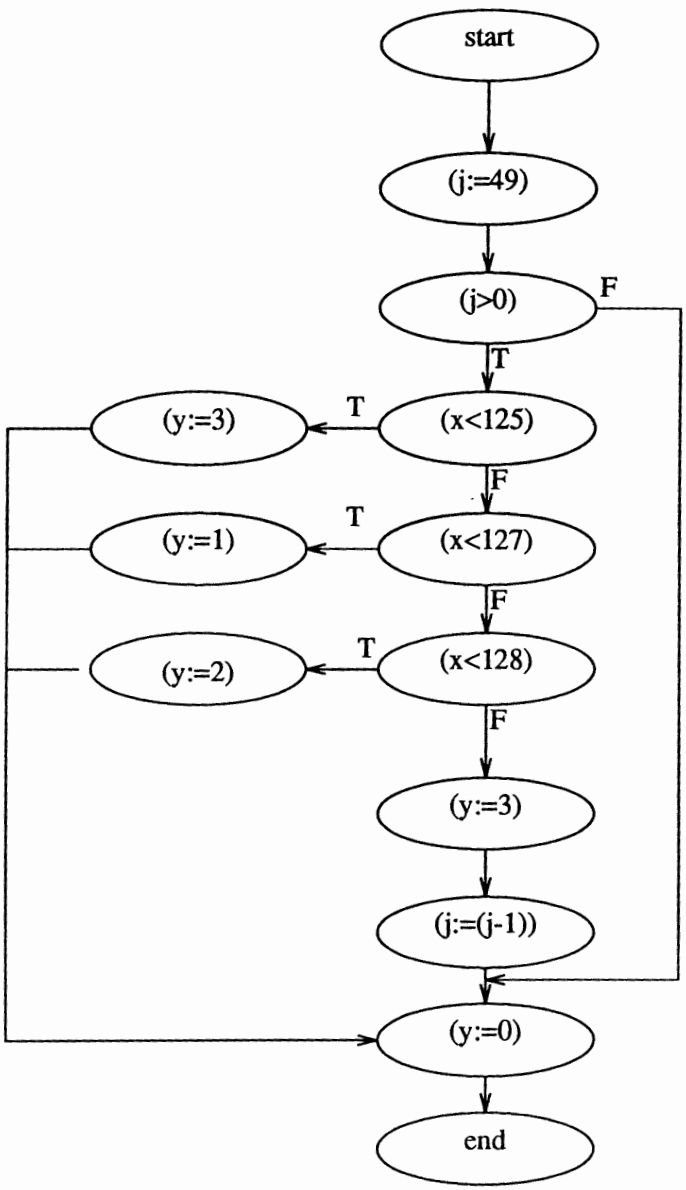

Figure 22. Flow graph for example 4.5.1.

Corresponding **GRAPH** format

(data

(coplisset

(1 (x 14 15)

((not 3) 3 14)

(x 13 3)

(x 5 13)

(x 7 13)

(x 9 13)

(x 11 13)

(x 12 13)

((not 10) 10 12)

(10 10 11)

((not 8) 8 10)

(8 8 9)

((not 6) 6 8)

(6 6 7)

((not 4) 4 6)

(4 4 5)

(3 3 4)

(x 2 3)

(x 1 2)))

(nolisset

(1 (stopadl 15 nil)

(14 14 nil)

```
   (13 13 nil)
   (5 12 nil)
   (7 11 nil)
   (cond 10 nil)
   (9 9 nil)
   (cond 8 nil)
   (7 7 nil)
   (cond 6 nil)
   (5 5 nil)
   (cond 4 nil)
   (cond 3 nil)
   (2 2 nil)
   (start 1 nil)))
(nalisset
 (1 (14 (:= y 0))
   (13 (:= j (plus j (minus 1))))
   (9 (:= y 1))
   (7 (:= y 2))
   (5 (:= y 3))
   (2 (:= j 49))))
(plisset
 (1 (10 (lessp x 128))
   (8 (lessp x 127))
   (6 (lessp x 126))
   (4 (lessp x 125))
```

```
    (3 (lessp 0 j))))
  (anlisset
   (1 (stopadl (15))
    (14 (14))
    (13 (13))
    (5 (5 12))
    (7 (7 11))
    (cond (3 4 6 8 10))
    (9 (9))
    (2 (2))
    (start (1))))
  (structlisset (1))
  (decset
   (1 adl
    c
    classification
    ((clock (1000)))
    (input ((x (j)) (p k1 8)))
    (intern (j (p k1 6)) (y (p k1 2)))
    (output (y (p k1 2)) (j (p k1 6)))))
  (bodset
   (1 (start)
    c
    (j := 49)
    (while (j > 0)
```

```
        do
        (if (x < 125) then (y := 3) (go 11))
        (if (x < 126) then (y := 2) (go 11))
        (if (x < 127) then (y := 1) (go 11))
        (if (x < 128) then (y := 2) else (y := 3))
        11
        (j := (j - 1)))
    (y := 0)
    (stopadl)))
(symlis (c 1))
(nnrset (1 16))
(lzmset (1 (y j)))
(indexlisset (1))
(asslisset (1))
(wlis 1)
( blc .
6 ))
```

The flow chart for the **GRAPH** output is shown in Figure 23.

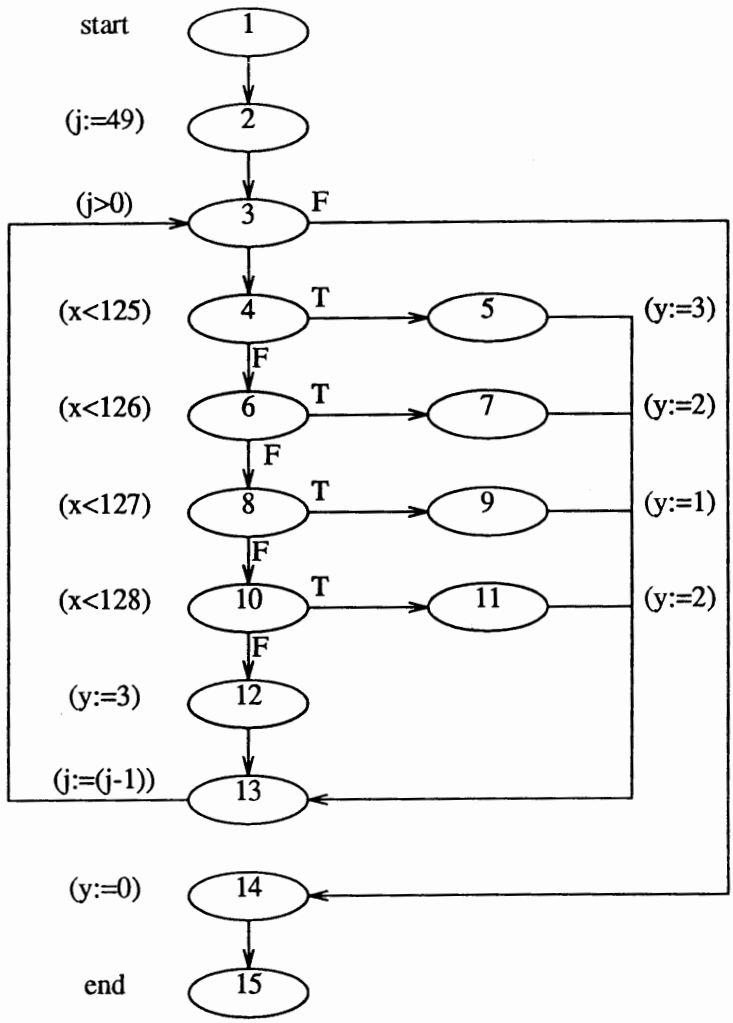

Figure 23. Flow graph for example 4.5.1.

CHAPTER V

STRUCT LANGUAGE

5.1 INTRODUCTION

The **STRUCT** language is composed of lists of nodes and arrows [12]. The node (abstract block) is a digital network with one or more inputs and one output. The arrows connect the outputs to the inputs of the various abstract blocks. There are two types of abstract blocks, functional blocks (F-type) and memory blocks (M-type). Functional blocks represent "combinational" logic functions since its outputs depend upon the present value of inputs. On the other hand, an M-type module is a "sequential" module, with an explicit clock input to control and updates. The arrows do not directly imply a particular input of a given node. If more than one arrow terminates on a particular input, a data selector is implied which selects one of the arrows to be connected with the input.

The **STRUCT** language consists of several lists which are:

VARLISTA - list of memory blocks (variables)

NODLISTA - list describing the nodes (abstract blocks)

COPLISTA - list describing the arrows (data transmission paths)

PLISOUT - list of predicates

NALISIMP - auxiliary list containing assignment statements together with the program number.

In each program, the lists are stored in the compound list VARLISTASET, NODLISTASET, COPLISTASET, PLISOUTSET and NALISIMPSET respectively.

5.2 SYNTAX OF VARLISTA.

The **"VARLISTA"** list is a list consisting of pairs of the form:

(NVAR NRN)

where:

NVAR - name of variable or parametric constant in the description.

NRN - number of a node corresponding to a given variable or parametric constant.

Example 5.2.1 - variables

VARLISTA = ((RP2 4) (RP1 3) (B 2) (A 1))

Describing four registers. Register RP2 is node 4.

5.3 SYNTAX OF NODLISTA

"NODLISTA" is a list of node descriptions. The list of nodes corresponds to list of abstract hardware blocks. There are two types of blocks, functional blocks (F-type) and memory blocks (M-type). Functional blocks represent "combinational" logic functions because the block output depends on the present value of input and is synchronous. An F-type block can range from a simple logic gate to a complex iterative circuit. On the other hand, a memory block is a sequential module, with an

explicit clock input to control loading. An M-type block is either a register or set of registers or ROM or RAM.

Each node has one of the following forms:

(NRN f FUNCT INP R1 C1 L1)

- Functional node

(NRN m FUNCT INP R1 C1 L1)

- Simple register node

(NRN m FUNCT INP R1 C1 L1 R2 C2 L2)

-Array register node

(NRN m FUNCT INP R1 ram LI (Words IND))

- Ram based array

(NRN m CONST INP R1 C1 L1)

- Simple constant

(NRN m CONST INP R1 C1 L1 R2 C2 L2)

- Indexed constant

(NRN m in () R1 C1 L1)

- input node

(NRN out FUNCT INP R1 C1 L1)

- output node

NRN - unique node number, assigned by IMPLEM

FUNCT - list of all operations that can be performed at this node.

INP - list of inputs to the node, the elements of which have the form:

((NRINP, NRC ... NRC) ....)

NRINP - internal name of the input for that type of node. Data inputs are

consecutively numbered from 1. Control input names include the following:

ad - address for RAM

rw - read/write

r - read only

w - write only

z - clear

lp - increment

lr -decrement

l - register load

sel - mux select

NRC - a number of an arrow reaching this input, corresponds to "COPLISTA"

R1 - type of data being output from a node

p - parallel array of logical bits

C1 - code of data being output from a node

k1 - standard binary data

k2 - gray coded data

L1 - number of bits being output from a node

R2 - type of data used for index or address

C2 - code of data used for index or address

L2 - number of bits in index or address

IN - special code for input nodes, considered a memory node

OUT - special code for output nodes

CONST - special code for constants. The value is contained in "COPLISTA" and the

value never changes

The input number "1" is the main input or the memory block. Input number "2" exists in the case of blocks corresponding to array variables and it is an address input. The rest of the inputs of the block are one bit inputs. The second form of the node description corresponds to the subscripted variable.

In the case of functional blocks, the succeeding inputs have succeeding numbers. Data transmitted into the inputs all have the same form.

Example 5.2.1 - of node descriptions:

(2 M IN NIL P K1 4)

This describes node 2 which is an input node (IN) with type M. There are no internal inputs (NIL). It is a 4-bit binary (K1) parallel (P) input.

(3 M (:= := := (Z :=)) ((L 8 10 4) (1 7 9 13) (Z 1)) P K1 4)

This describes node 3 which is a register (M-type) node. It can be zeroed by the control unit ((Z :=)) with arrow connected to input Z(Z 1). The register will be loaded from input 1 when corresponding L input is activated by the control unit. For example, if arrow 8 is active then the data on arrow 7 will be loaded into the register via input 1. Input 1 is a 4-bit parallel binary input.

(5 F (PLUS) ((1 11) (2 12)) p K1 5)

This describes node 5 which is a function (F-type) node. It performs the addition of data on arrow 11 and 12 applied to input 1 and 2 respectively. Both inputs are 5-bit parallel binary inputs.

5.4 SYNTAX OF COPLISTA

The list **"COPLISTA"** is a list of arrows. Arrows are data transmission paths between blocks and they are implemented as wires between blocks. The description of an arrow has one of the following forms:

(NRC NRN (DZO GZO) (DZI GZI) R C)

- standard data connection

(NRC (vect VALUE) (DZO GZO) (DZI GZI) R C)

- a vector or constant value

(NRC s S)

- set of control signals from a control unit.

NRC - unique arrow number assigned by IMPLEM

NRN - the node number from where this arrow originates or the output of node number NRN

VALUE - the binary value of a constant or vector, in the form of 1s and 0s with no leading 0s

DZO - the lower bit position from the origination node

GZO - the upper bit position from the origination node

DZI - the lower bit position at the node where the arrow terminates

GZI - the upper bit position at the node where the arrow terminates

R - A code describing the type of information conveyed by the arrow

p - parallel data

d - 1 bit logical value

C - a code describing the representation of the transmitted information

S - The number of control signals from the control unit

The value of DZO, GZO, DZI, and GZI allow the selection of a specific range of bits from or to a node.

Example 5.4.1 - arrow descriptions:

(9 3 (1 3) (0 2) P K1)

This describes arrow number 9 which leaves node 3 and connects output bits 1 to 3 of node 3 to input bits 0 to 2 of the destination node. The bits are in parallel binary format.

(21 (VECT (0)) (0 0) (0 3) P K1)

This describes arrow number 21 which connects the one bit vector constant 0 from output 0 to input 0 of the destination node. Inputs 1 to 3 of the destination node default to zero since they are unused.

(1 S)

This describes arrow number 1 which comes from the external control unit.

The control signal arrow stands for the transmission of control signals into the appropriate control input of memory and certain functional blocks.

5.5 SYNTAX OF PLISOUT

The **"PLISOUT"** list is the description of outputs from conditional predicates. Elements of this list are the predicates and have the following form:

(NRP PLIS NRC)

where:

NRP - number of a predicate from plislist.

PLIS - The contents of a predicate from plislist.

NRC - The number of an arrow transferring the value of the given predicate.

Example 5.5.1 - description of PLISOUT

(5 (EQUAL RP2 9) 22)

This describes predicate number 5 which compares variable RP2 to the constant zero. The result is sent to the control unit via arrow 22.

(7 (AND (LESSP B 4) (EQUAL B C)) 5)

This describes predicate number 7 which checks for the condition where both predicate B is less than 4 and predicate B is equal to variable C. The result is sent to the control unit via arrow 5.

5.6 SYNTAX OF NALISIMP

The **"NALISIMP"** list is an auxiliary list indicating all operations taking place in the implementation of the machine, the elements of which have the following form:

(NRNVAL NALIS ((DRNC GRNC)))

- operations involving simple registers with one input

(NRNVAL NALIS ((DRNC GRNC) ((CONTROL NRC)))

where:

NRNAL - number of assignment statement from NALIS list.

NALIS - contents of assignment statement from NALIS list.

DNRC & GNRC - scope of numbers of arrows included in the abstract implementation of the given statement.

NRC - arrow number for control signal.

CONTROL - symbol indicating the type of control signal used for this operation. These symbols include the following:

ad - address for RAM

rw - read/ write

sel - mux select

NALISIMP list allows fast searching of arrows included in abstract implementation of the given assignment statement. It may be useful for optimization and implementation of digital systems gating the inputs to abstract blocks.

Example 5.6.1 - an element of the NALISIMP list

(4(:= RP2 B) (15 16))

This describes assignment statement 4 from the NALIS list. The assignment statement is (RP2 := B) and requires arrows 15 and 16.

(9(:= RP1 (PLUS RP1 A)) (11 14))

This describes assignment statement 9 from the NALIS list. The assignment statement is (RP1 := (RP1 + A) and requires arrows 11, 12, 13 and 14.

Example 5.6.2 General examples

Description of arrows

(1 1 (0 3) P K1)

(2 (VECT (110)) (0 2) (0 2) P K1)

Description of node

(3 2 (0 4) (0 4) P K1)

(2 F (PLUS) ((1 1) (2 2)) P K1 4)

In the above example node 1 corresponds to A which is a four bit parallel variable with code K1 and node number 2 corresponds to constant number 6, The implementation of (PLUS A 6) is shown above.

The graphical representation of the above example is shown in Figure 24.

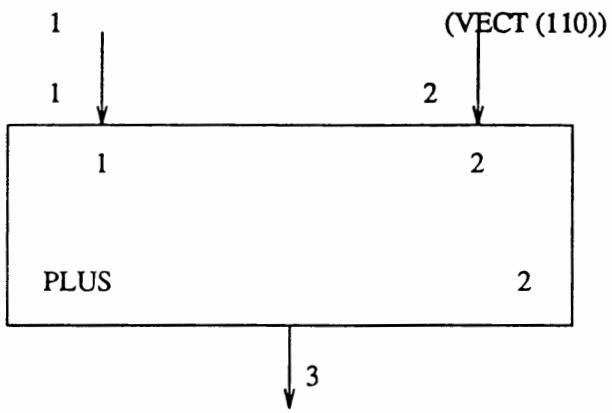

Figure 24. Graphical representation of example 5.6.2.

Description of arrows

(1 1(0 3) (0 3) P K1)

(2 (VECT (0)) (0 0) (0 0) P K1)

(3 3 (0 0) (0 0) P K1)

(4 (VECT (1 1)) (0 1) (0 1) P K1)

Description of nodes

(2 M (:=) ((1 4) (L 3)) (P K1 4)

(3 F (EQUAL) ((1 1) (2 2)) P K1 4)

In the above example node 1 corresponds to Variable A, and node 2 corresponds to variable B. It is for the statement (IF (EQUAL A 0) THEN (:= B 3))

The graphical representation of the above example is shown in Figure 25.

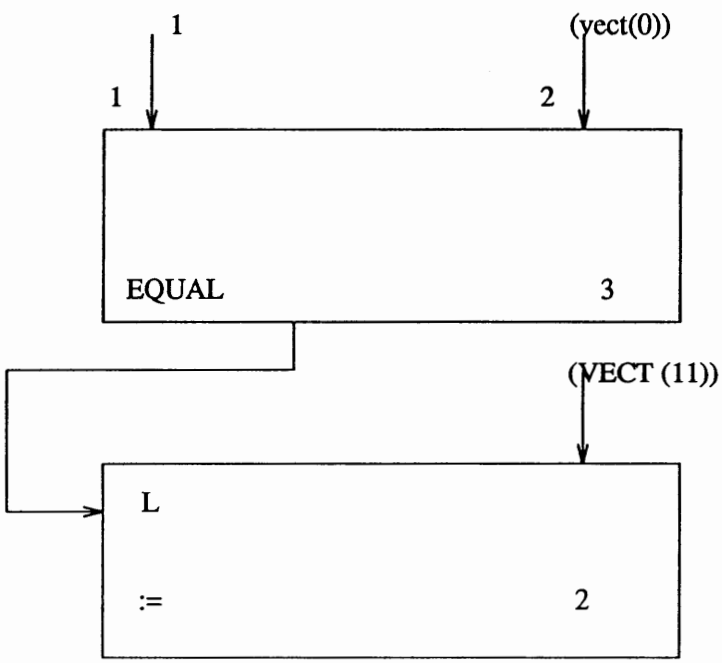

Figure 25. Graphical representation of example 5.6.2.

Example 5.6.3 - GRAPH file and its converted STRUCT form.

GRAPH format

(data

(coplisset

(1 (x 3 4)

(x 2 3)

(x 1 2)))

```
(nolisset
 (1 (stopadl 4 nil)
   (3 3 nil)
   (2 2 nil)
   (start 1 nil)))

(nalisset
 (1 (3 (:= h ( not f g))
 (2 (:= e (and a d))))
(plisset (1))
(anlisset (1 (stopadl (4)) (3 (3)) (2 (2)) (start (1))))
(structlisset (1))
(decset
 (1 adl
   c
   classification
   ((clock (1000)))
  (input (a (d)) (b (d)) (f (d)) (g (d)))
  (output (e (d)) (h (d)))))

(bodset
(1 (start)
  c
(e := a and d) (h := f not g)
(stopadl)))
(symlis (c 1))
```

(nnrset (1 5))

(lzmset (1 (h e)))

(indexlisset (1))

(asslisset (1))

(wlis 1)

(blc . 6))

The flow chart for the above **GRAPH** file is shown in Figure 26.

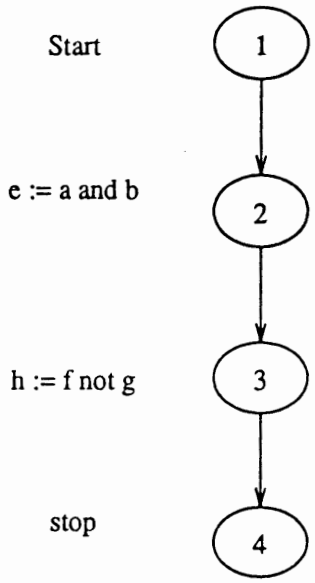

Figure 26. Flow graph for example 5.6.3.

The corresponding **STRUCT** format

((

(input (a (d)) (b (d)) (f (d)) (g (d)))

nil

(output (e (d)) (h (d)))

```
nil
)
(nodlista (5 out (:=) ((l 9) (1 8)) p kl 1)
(6 out (:=) ((l 7) (1 6)) p kl 1)
(4 m in nil p kl 1) (3 m in nil p kl 1) (2 m in nil p kl 1)
(1 m in nil p kl 1))

(coplista (9 s 1) (8 1 (0 0) (0 0) p kl)
(7 s 1) (6 3 (0 0) (0 0) p kl))

(varlista (h 6) (e 5) (g 4) (f 3) (b 2) (a 1))
(plisout)
(nalisimp (2 (:= e (a and b)) ((8 9)))
        (3 (:= h (f not g)) ((6 7))))
)
```

The graphical representation for the above **STRUCT** file is shown in Figure 27. In the Figure all the numbers are node numbers.

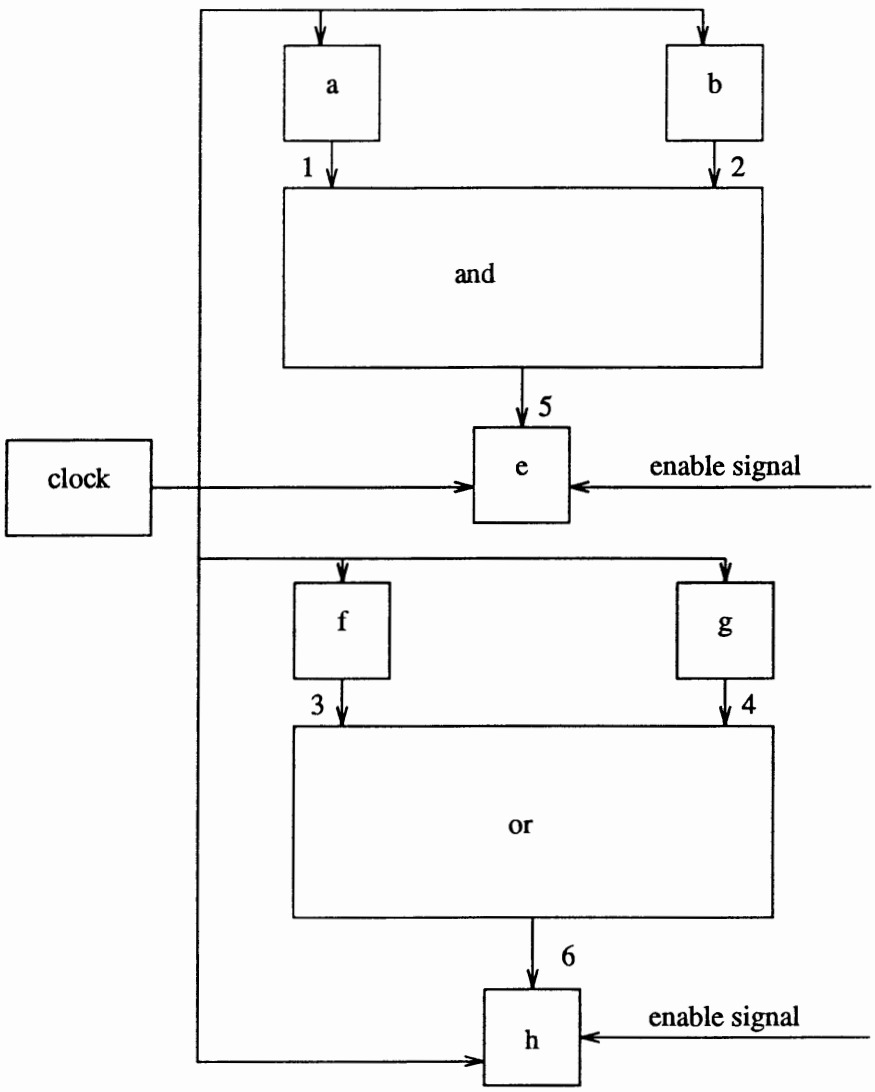

Figure 27. Graphical representation of the STRUCT file.

CHAPTER VI

VHDL TO ADL COMPARISON

6.1 INTRODUCTION

VHDL is a hardware description language used to document an electronic system design created in late eighties and being currently an industry standard [13]. A **VHDL** design consists of several separate design units, each of which is compiled and saved in a library. The four main design units are: entity, architecture, configuration, and package. The interface signals for the design are described in entity. The design's behavior is specified in an architecture. A configuration selects a variation of a design from a library. For convenience, certain frequently used specifications can be stored together in a package [14]. For description of **ADL** refer to chapter 3.

Example 6.1.1 A VHDL program

```
package exam is
  constant unit_delay : time
```

Package

```
end exam

entity compare is
port ( x,y : in bit;
```

Entity

```
       z: out bit);
```

end compare;

library mgc_library;

Architecture

Use mgc_library.exam.all;

architecture gate of compare is

begin

c <= NOT (a XOR b) after unit_delay;

end gate;

In the above example

package exam - Provides a sharable constant.

Entity compare - Names the design and signal ports.

Architecture gate of compare - Provides details of the design.

A configuration of compare - Designates flow as the latest compiled architecture.

6.2 THE FEATURES THAT ARE IN ADL BUT NOT IN VHDL

6.2.A Identities

An **identity** is a symbol or name identical to some value, variable, or expression. Different names for the same variable can be used in **ADL. Identities** are declared in the **identity** list.

Example 6.2.A.1 - Identities

In this example "a" and "b" are declared identical. "e" and the range of bits 1 through 4 in array d are identical.

(iden (a b) (e (d [1 % 4])))

6.2.B SYMB List

A single statement can be given a symbolic name in ADL. The symbolic name can be used in place of the statement in the program. Use of symbolic names reduces the amount of repetitive typing in some programs. In the **iden** list only a single name was used to represent the identity. In the **symb** list, a whole expression or statment can be used for the symbol. Symbolic names and their statements are declared in the **symb** list.

Example 6.2.B.1 - Symbolic List

(symb (c1 (k := (k + 1)))
((a := (c + b)) (adder1 (a c b))))

In the first example, the c1 symbol would be replaced by the statement (k := (k + 1)) in the program. In the second example, the statement (a := (c + b)) is replaced by the subroutine call (adder1 (a c b)). The subroutine could be a structural description of a special adder.

6.2.C Labels

Statements can have **labels. Labels** are used to mark the destinations of " **go** " statements. " **go** " statements are similar to " **goto** " statements in BASIC. A **label** is

always in numeric and alphabets are not permitted. A **label** is placed on the same line as the statement it refers to, but outside the parentheses for that statement. A **label** is inside of any parenthesis surrounding a block of statements. Each labeled statement must have a different **label** number.

Example 6.2.C - Statements with Labels

```
10    (a := 5)
20    (if (a < 5) then (a := (a + 1))
                    (go 20)
              else (a := 100)
                    (go 30)
                    )
30  (c := 5)
```

6.2.D Go Statements

Go statements are used to unconditionally branch to a labeled statement. **Go** statements and their corresponding destination statements can occur anywhere in the program. Jumps out of **if-then else** statements and **while** loops are allowed. The "value" of a **label** cannot be created using an expression. For example: (i := 99) and (go i) is invalid. The basic grammar for the **go** statement is:

<go statement> ::= (go <label>) ;

6.2.E Cond Statement

The **cond** statement is an extension of the **if-then-else** statement. Within the **cond** statement can be several branches, but only one is executed. Each branch has its own predicate controlling execution of that branch. When a **cond** statement is encountered, the predicate of the first branch is executed. If it is true then that branch is executed and program execution continues with the next statement after the **cond** statement. If the first predicate is false then the next branch is executed. This process goes on until one of the branches is executed or the end of the **cond** statement is reached. There is no "default" branch. There must be at least two branches in a **cond** statement. Jumps out of a **cond** statement using the " **go** " statement are allowed.

Refer example 3.10.D.1 for example on **cond** statement.

6.2.F Set and Reset Statements

Set and **reset** statements control the turning on and off of many one-bit signals. **Set** means an output variable is 1 and **reset** means an output variable is 0. Only logical variables can be **set** and **reset** because they are one bit signals. Once a variable is **set** or **reset,** it holds that value until the variable is changed again.

Example 6.2.F.1 - Set and Reset

(set a b c)

(reset x y z)

6.2.G Parallel Execution Statements

In **ADL** the user can design a system with parallel operations. Parallel

operations use the same control unit but multiple hardware elements are enabled at the same time. An **ADL** program can be written so that individual statements or blocks of statements operate in parallel. Multiple statements can be directed to execute in the same cycle. An algorithm can be split into parallel blocks of statements. Control statements join parallel blocks back into one process. Parallel operation is done by the instruction **fork,** it is explained in chapter 3, section 3.12. Multiple execution statement is done by the instruction **sim** , it is explained in chapter 3, section 3.11.E.

6.3 THE FEATURES THAT ARE IN VHDL NOT IN ADL.

6.3.A process statement

The **process** statement is a concurrent statement that defines the scope of each **process** The **process** statement defines a specific behavior to be executed when that **process** becomes active [15],[16]. The syntax is:

process_label:
process
declarations
begin
statements
end process;

The process-label defines a named label for the **process.** The declarations section defines the local data environment needed by the **process.** The statements section of the **process** is the sequential program which defines the behavior of the

process. Each **process** statement defines a specific action, or behavior, to be performed when the value of one of its sensitivity signals changes. This action is defined by the sequentially ordered execution statements in the process.

Example 6.3.A.1 - Process statement

```
Entity lo_w is
  port(a,b,c: in integer);
end lo_w;

architecture behave of lo_w is
begin

process
variable low: integer := 0;
begin

        wait on a,b,c;
        if a < b then low := a;
        else low := b;
        end if;
        if c < low then
        low := c;
        end if
end process;

end behave;
```

6.3.B Case Statement

The **case** statement is a form of conditional control provided in **VHDL.** The **case** statement is used to select a collection of statements based on the range of values of a given expression. The designer gives the expression and identifies a collection of expressions for each possible value of the type of the expression.

Example 6.3.B.1 - Case Statement

```
process
begin
    case selector is
        When "00" =>
         output <= in0;
        when "01" =>
         output <= in1;
        when "10" =>
         output <= in1;
        when "11" =>
         output <= in3;
    end case;
    wait on selector;
end process;
```

The **when** conditions are called the arms of the **case** statement. There may be any number of these, but no two arms can have the same value and every value in the range of the type must be represented.

6.3.C Next Statement

The **next** statement skips execution to the **next** iteration of an enclosing loop statement. The syntax for the **next** statement is:

next [loop_label] [when condition];

The **loop-label** and **when** condition are both optional. When this statement is executed, control goes to the bottom of the loop identified by the label and the loop is begun again.

Example 6.3.C.1 - Next statement

```
For i in 0 to 10 loop
        if (i = 5) then
        next;
        end if;
end loop;
```

6.3.D Exit Statement

The **exit** statement completes the execution of an enclosing loop statement. The **exit** statement has two general forms:

exit loop_label;

exit loop_label when condition;

The first form exits the enclosing loop with the given loop label. The second form does the same thing but only if the condition is true.

Example 6.3.D.1 - Exit Statement

For i in 0 to 10 loop

if (i = 5) then

exit;

end if;

end loop;

6.3.E Assert Statement

The **assert** statement checks to determine if a specified condition is true, and displays a message if the condition is false. The syntax is:

assertion_statement

assert condition

[report expression]

[severity expression];

assert writes out text messages during simulation. There are four levels of severity: failure, error, warning, note. The **assert** statement is useful for timing checks, out-of-range condition, etc.

6.3.F Enumerated Types

The **Enumerated** type declaration lists a set of names or values defining a new type. The syntax is:

enumeration_type_definition

type identifier (enumeration_literal {, enumeration_literal});

enumeration_literal

identifier

character_literal

This feature allows the user to declare a new type using character literals or identifiers.

Example 6.3.F.1 Enumerated Types

type tools is (hammer, saw, drill, wrench);

6.4 Semantic difference between ADL and VHDL

In **VHDL** its design is always running in a simulated time, and events occur in successive time steps. **VHDL** has concurrency and component netlisting, which are not present in **ADL.** **VHDL** supports concurrency using the concept of concurrent statements running in simulated time. In **VHDL** , the design hierarchy is accomplished by separately compiling components that are in a higher-level component. **ADL** is like a programming language where as **VHDL** is a full form hardware description language. Parallelism in **ADL** is obtained easily where as it is difficult in **VHDL**

6.5 WHY VHDL WAS CHOSEN

VHDL is the most used hardware description language in industries. The industries use **VHDL** for the following reasons

1.

VHDL allows the user to design, model, and test a system from a high level of abstraction down to the structural gate level.

2.

VHDL descriptions created following by **VHDL** synthesis guidelines can be run through a synthesis tool to create gate-level implementations of designs.

3.

The **VHDL** tool that is available at PSU is integrated into one overall design environment from Mentor Graphics. It is possible to do a system-level simulation mixing high-level abstract descriptions with detailed gate-level models.

4.

VHDL supports top-down design methodology, describing the behavior of the high-level blocks, analyzing them, and refining the high-level functionality as required before reaching the lower abstraction levels of design implementation.

5.

VHDL supports modularity, the principle of partitioning a hardware design and the associated **VHDL** description into smaller units.

6.

VHDL supports abstraction, which is grouping details that describe the function of a design unit but do not describe how the design unit is implemented.

7.

VHDL supports information-hiding, by which the implementation details of one module is hidden from other modules.

Because of the above mentioned properties, **VHDL** is a very good language for detailed hardware specification, and it is commonly used in logic synthesis, technology mapping and layout systems [17]. On the other hand **ADL** allows for very easy description of "program-like" parallel and sequential behavior of hardware. In this respect it is much closer to such specification languages as C,

PASCAL, microprogrammed notation and VERILOG. The **DIADES** system includes several system-level and high-level transformations that are missing in the Mentor tools. On the other hand, Mentor tools have many sophisticated Logic Synthesis, physical design, Simulation or FPGA related programs that are absent from **DIADES.** Integrating these two large systems allows to create a very comprehensive and complete system of system/high level/logic/technology/physical design that is not comparable to anything that exists in industry. In the next chapters the translation from **DIADES** internal format to Mentor's **VHDL** will be presented in all necessary detail. Integration of the two systems allows us to compare various variants of design where different trade-offs between **DIADES** designed and Mentor designed design phases are considered.

CHAPTER VII

COMPILATION OF GRAPH LANGUAGE TO VHDL

7.1 INTRODUCTION

Compilation of **GRAPH** format to **VHDL** is done by program *gvhdl*. This program translate the descriptions in the **GRAPH** format to a **VHDL** description. Input data to the program is a description of the system under design in the form of familiar **GRAPH** language lists stored in the compound lists: COPLISSET, NOLISSET, NALISSET, PLISSET, ANLISSET, and STRUCTLISSET. Output data from the program will be the corresponding **VHDL** description. The program uses the information in the various lists in the **GRAPH** format to create the **VHDL** description.

7.2 CONVERSION OF GRAPH TO VHDL

The program is divided into different subprograms: *nali.c, noli.c, cop.c, pli.c and loop.c* . The subprogram *entity.c* creates the entity information for the **VHDL** description from the decset which is present in **GRAPH** file. The decset has the input, output, and intern informations of the digital system. The input will be transformed as input port and the output as the output port in the entity declaration in the VHDL description. The intern which is basically like a temporary variable will be declared in the architecture part of the **VHDL** description, as a variable.

Example - conversion of entity

Graph input output declaration

(input (a(d))(b(d)))

(output (c(d)))

Corresponding entity declaration

entity clas is

port(

a,b: in bit;

d: out bit);

end clas;

The subprogram *cop.c* accesses the COPLISET list in the **GRAPH** file and creates two intermediate files *pli* and *nal*. The COPLISET list in the **GRAPH** file represents arrows. The arrows indicate how control flows through the **GRAPH.** The created intermediate file *nal* has all the state change information. The state change information has the node numbers which need state change. If the arrow in the COPLISET has "x" as the control transfer type then the second item in the COPLISET which is a node number that needs a state change. The program writes the node numbers of the arrows which has "x" as the control transfer type into the *nal* file. The *pli* file has the conditional control information. The arrows which have a node number as the control transfer type are the arrows which represent comparison operation. If a comparison result is false, then the arrow corresponding to the node number which has keyword will be the destination node. The program writes the node numbers of the arrows which have the keyword *not* into the *pli* file.

Example - 7.2

(coplisset

(x 1 2)

(x 2 3)

(11 11 12)

((not 11) 11 13)

The *pli* file will have

11 13

It means that 11 is a conditional control node and if the result of the comparison node is false then the control jumps to node 13.

The *nal* file will have

1 2

2 3

In the above file if we consider line 1 which is "1 2", it means there is an unconditional control transfer between node 1 and node 2. If there is an control transfer between two nodes then in the corresponding **VHDL** description it will be reflected as a state change.

The program *noli.c* accesses the NOLISSET list in the **GRAPH** file and plays an important role in the creation of the **VHDL** file. The NOLISSET list describes properties of each node. The nodes that are not described in other lists are described in NOLISSET. The NOLISSET also has the sequence of the nodes. The program *noli.c* accesses each line in the NOLISSET starting from the line which has node 1 and accesses the other lines according to the sequence of the nodes. Since each nodes properties are described in NOLISSET, the program can find out whether the node is an assignment node, a conditional node, or an equivalent node.

Example 7.3

(nolisset

(7 11 nil)

(cond 10 nil)

(9 9 nil)

(start 1 nil)

The first item in the NOLISSET is the property. The second item is a node number, which has that specific property. If the first item is the keyword then it is a comparison node, if it has the keyword then it is a start node. If the first item is a node number as in the second item, then it is an assignment node. If the first item is a node number and the second item is a different node number then it is an equivalent node. In the above example (start 1 nil) means the state machine starts with node 1. (9 9 nil) means 9 is an assignment type node. (cond 10 nil) means it is an comparison node. (7 11 nil) means it is an equivalent node, where nodes 7 and 11 are assignment type nodes and node 11 is equivalent to node 7.

The program after finding the property of the node calls either one of the programs *nali.c* or *pli.c*. If the node is of assignment type then the program *noli.c* calls the program *nali.c*. If the node is of comparison type then the program *noli.c* calls the program *pli.c*. If the node is of equivalent type then the program, depending on the type of the first item in the NOLISSET which is a node number, calls either *nali.c* of *pli.c*. The program *nali.c* accesses the NALISSET list. The NALISSET list describes all of the assignment type operations. Each operation consists of a assignment type, a destination for the results of the operation, and an expression. The first item of a NALISSET will be a node number.

When the program *noli.c* calls the program *nali.c*, it also sends the node numbers to *nali.c* in a sequence as in the NOLISSET. Then the program *nali.c* takes this node number and accesses the NALISSET. From the NALLISSET it finds the assignment statement corresponding to that node number and prints out the **VHDL** format in the output file. The program just matches the given node number with the first item in the NALISSET, and finds the corresponding assignment statement. From the assignment statement, the program checks for the type of the destination. Then the program looks into the assignment operator and finds the equivalent of it in the **VHDL** format. Once all these information is obtained the program prints the **VHDL** format of the assignment statement in the output file.

Example 7.4

(nalisset

(2 (:= j 49))

(7 (:= y 2))

(13 (:= j (plus j (minus 1))))

From the above example, if the program *nali.c* is called and given the node number 2 then it will take the line (2 (:= j 49)) and will convert it into j <= 49 in **VHDL** format. If it is given the node number 13 then it will take the line (13 (:= j (plus j (minus 1)))) and will convert it into j <= j - 1 in the **VHDL** format.

The program *pli.c* accesses the PLISSET. The PLISSET list contains the node descriptions for comparison operations. Each operation consists of an predicate, a destination for results of the operation, a relation operator, and an expression. The first item in the PLISSET will be a node number. When the program *noli.c* calls the program *pli.c* it send a node number according to the sequence of the NOLISSET.

The program *pli.c* gets the node number and finds the corresponding node descriptions from the PLISSET and the comparison operation for that particular node. Then program *pli.c* finds the comparison type by calling the program *loop.c*.

The program *loop.c* accesses the BODSET which has the **ADL** description of the **GRAPH** file. When the program *pli.c* calls the program *loop.c*, it also sends the conditional statement expression along. The program *loop.c* matches this given expression with the conditional statement expression in the BODSET, finds the corresponding type of comparison operation, and sends it back to the program *pli.c*. Once the program *pli.c* gets the type of the comparison operation it prints the corresponding **VHDL** format in the output file.

Example 7.5

(plisset
(3 (lessp 0 j))))
(6 (lessp x 126))
(8 (lessp x 127))

Let us assume that the above PLISSET is an argument of program *pli.c*. When node number 3 is passed to *pli.c*, and a statement *(while (j > 0))* is included in the BODSET list, then the **VHDL** format printed in the output file will be *while(j > 0)*. If the given node number is 8 and the BODSET has the statement *(if (x < 127))*, then the **VHDL** format printed in the output file will be *if(x < 127)then*.

Before calling the programs *nal.c* and *pli.c* the program *noli.c* checks the node number with the intermediate files *pli* and *nal*. The intermediate file *pli* will have the info about the end of conditional control statements. If the node number is present in

the *pli* file then the program prints the end of conditional control info on the output file. The intermediate file *nal* has the state change information in it. If the node number is present in the file *nal* then the program prints the state change information in the output file according to the **VHDL** format.

Example - 7.6

Graph *file*

(data

(coplisset

(1 (x 3 4)

(x 2 3)

(x 1 2)))

(nolisset

(1 (stopadl 4 nil)

(3 3 nil)

(2 2 nil)

(start 1 nil)))

(nalisset

(1 (3 (:= h (f or g))

(2 (:= e (and a b))))

(plisset (1))

(anlisset (1 (stopadl (4)) (3 (3)) (2 (2)) (start (1))))

(structlisset (1))

```
(decset
 (1 adl
   c
   classification
   ((clock (1000)))
 (input (a (d)) (b (d)) (f (d)) (g (d)))
 (output (e (d)) (h (d)))))

(bodset
(1 (start)
(e := a and d) (h := f not g)
(stopadl)))
(symlis (c 1))
(nnrset (1 5))
(lzmset (1 (h e)))
(indexlisset (1))
(asslisset (1))
(wlis 1)
( blc . 6 ))
```

The flow chart for the above **GRAPH** file is shown in Figure 28.

The corresponding **VHDL** description

```
entity clas is
port(
  CLOCK:in bit;
```

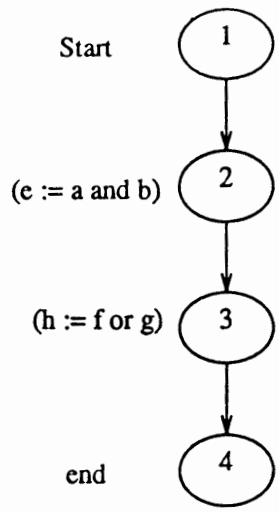

Figure 28. Flow graph for example 7.6.

```
  a,b,f,g:in bit ;
  e,h:out bit);
end clas;

architecture adl1 of clas is
type state_type is (s0,s1,s2,s3);
signal current_state,next_state:state_type;
begin

pro1:process
begin

wait until CLOCK'event and CLOCK='1';
current_state <= next_state;
end process pro1;

pro2:
```

```
process(current_state,a,b,f,g)
begin

next_state <= s0;
case current_state is
when s0=>
next_state <= s1;
when s1=>
e <= a and b;

next_state <= s2;
when s2=>
h <= f or g;

next_state <= s3;
when s3=>
end case;
end process pro2;
end adl1;
```

CHAPTER VIII

COMPILATION OF STRUCT LANGUAGE TO VHDL

8.1 INTRODUCTION

Programs *svhdl* and *mvhdl* translate a **STRUCT** format description of a digital system into a **VHDL** format description. The program *svhdl* creates a **VHDL** code as it is described in the **STRUCT** format. The program *mvhdl* creates a **VHDL** code in which all the operations are implemented in gate level. **STRUCT** format is an abstract block description of a digital hardware structure. A block with function and size information represents each hardware operation. Abstract arrows with size and transmitted information type represent data transmission paths between blocks. There are inputs and outputs for communicating with the outside world. Input data to the program is in the form of **STRUCT** language lists stored in the compound lists: VARLISTA, NODLISTA, COPLISTA, PLISOUT, and NALISIMP. Output data from the program will be the corresponding **VHDL** description. The program uses the information in the various lists in the **STRUCT** format description to get the **VHDL** description.

8.2 CONVERSION OF STRUCT TO VHDL

The program *svhdl* is divided into different subprograms: *port.c, nod.c nali.c, pli.c*. The program *port.c* creates the entity information for the output **VHDL** file.

The program *port.c* accesses the input and output information described in the **STRUCT** file and transforms it into corresponding input port and output port. The input and output port information in the **STRUCT** file will be in the declaration section. For each input list, output list, and constant list in the declaration section of the **STRUCT** file a node is created in the NODLISTA set.

The converted port information will be written in the entity section of the output **VHDL** file by the program.

Example 8.2.1 - Conversion of entity

STRUCT input output declaration

(input (a(d)) (b (d)))

(output (c(d)))

Corresponding entity declaration

```
entity clas is
port(
  a,b: in bit;
  c: out bit);
end clas;
```

The program *nod.c* comes into effect once the entity information is created. The program *nod.c* accesses the NALISIMP list and PLISOUT list in the **STRUCT** file. The NALISIMP list is an auxiliary list indicating all the operations taking place in the implementation of the machine, it has the information of the assignment statements. For more information on NALISIMP list refer to section 5.5. The first item of the

NALISIMP list is a number of an assignment statement. The PLISOUT is a list of predicates and corresponding arrows, it has the information of the conditional statements. For more information on PLISOUT list refer to section 5.4. The first item of the PLISOUT list will be a number of a predicate.

The program *nod.c* when accessing the lists NALISIMP and PLISOUT creates an intermediate file called *nal*. The *nal* file will have the first items of the NALISIMP list and PLISOUT list. When the program writes the first item of the NALISIMP list in the *nal* file it adds a letter "a" to it, when it writes the first item of the PLISOUT list it adds a letter "c" to it.

Example 8.2.2 Creation of the nal file

(nalisimp
(2 (: j 49) ((sel 2) (14 15)))
(5 (: y 3) ((sel 4) (12 13)))
(7 (: y 2) ((sel 4) (10 11)))

(Plisout
(3 (lessp 0 j) 30)
(4 (lessp x 125) 27)
(6 (lessp x 126) 24)

The *nal* file created by the program *nod.c*, when it accesses the NALISIMP list and PLISOUT list in the above example, is shown below.

2 a
5 a

7 a

3 c

4 c

6 c

The program first writes the assignment statement numbers from the NALISIMP list and then it writes the predicate numbers from the PLISOUT list. Then the program *nod.c* rearranges the contents of the *nal* file in an ascending order. The program accesses the contents of the *nal* file line by line and it scans for the first item in the line which is a number and then it arranges it in ascending order. The rearranged *nal* file is shown below.

2 a

3 c

4 c

5 a

6 c

7 a

The program *nod.c* then accesses the rearranged *nal* file one line at a time starting from the first line and calls one of the programs *nali.c* or *pli.c*. If the line in the *nal* file has the letter "a" then it calls the program *nali.c* and if the line has the letter "c" it calls the program *pli.c*. The program *nali.c* accesses NODLISTA list, COPLISTA list and NALISIMP list. The program *nali.c* is used to implement the assignment operations. The program *pli.c* accesses NODLISTA list, COPLISTA list and PLISOUT list. The program *pli.c* is used to implement conditional control

statements. The NODLISTA list is a list of node descriptions. For more information on NODLISTA list refer to section 5.3. The COPLISTA list is a list of arrows. For more information on COPLISTA refer to section 5.4. The programs *nali.c* and *pli.c* gets the node and arrow information from the NODLISTA list and COPLISTA list, then the programs accesses the NALISIMP list if it is *nali.c* or PLISOUT list if it is *pli.c* and writes the **VHDL** code in the output file.

The program *nali.c* implements the addition operations, subtraction operations, multiplication operations using shift registers. The addition operation is done using two shift registers. The two shift registers will hold the numbers to be added. Both the shift registers are connected to a common clock. At each clock time, the shift registers give out one bit. The bits given out by the shift register is fed to a full adder. The output of the full adder is given to an shift register which holds the result. This whole addition operation is written as an component in **VHDL** and it is called in the main **VHDL** program when it is needed. Subtraction operation is done in the same way as addition operation by adding the 2's complement of the number to be subtracted. The block digram of the addition operation is shown in Figure 29.

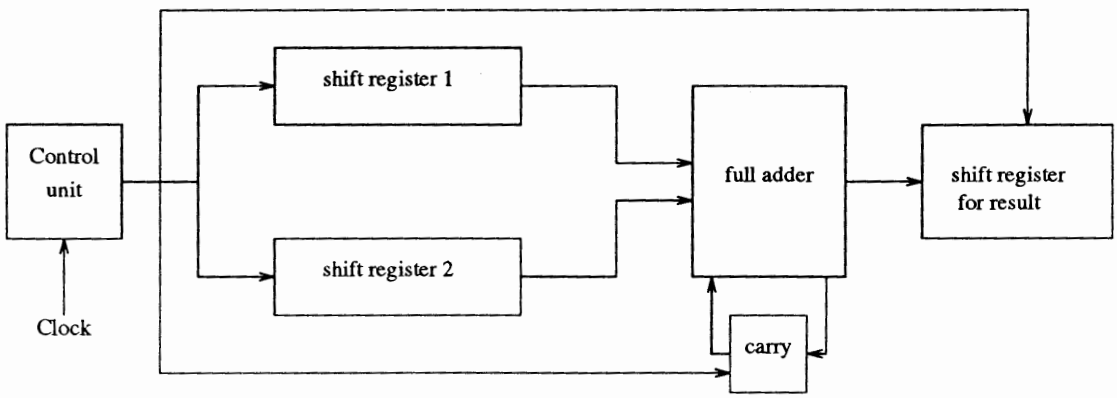

Figure 29. Addition operation.

The multiplication operation is implemented using two registers. The multiplier is stored in a shift register and the multiplicand is stored in an register. Both the registers are connected to an AND block. For each clock time the shift register gives out one bit. The bit given out by the shift register is fed to the AND block which already has all the multiplicant bits. The output of the AND block is fed to an adder and result holder block. This whole multiplication operation is written as an component as addition operation and it is incorporated in the main **VHDL** program when it is needed. The block diagram of the multiplication operation is shown in Figure 30.

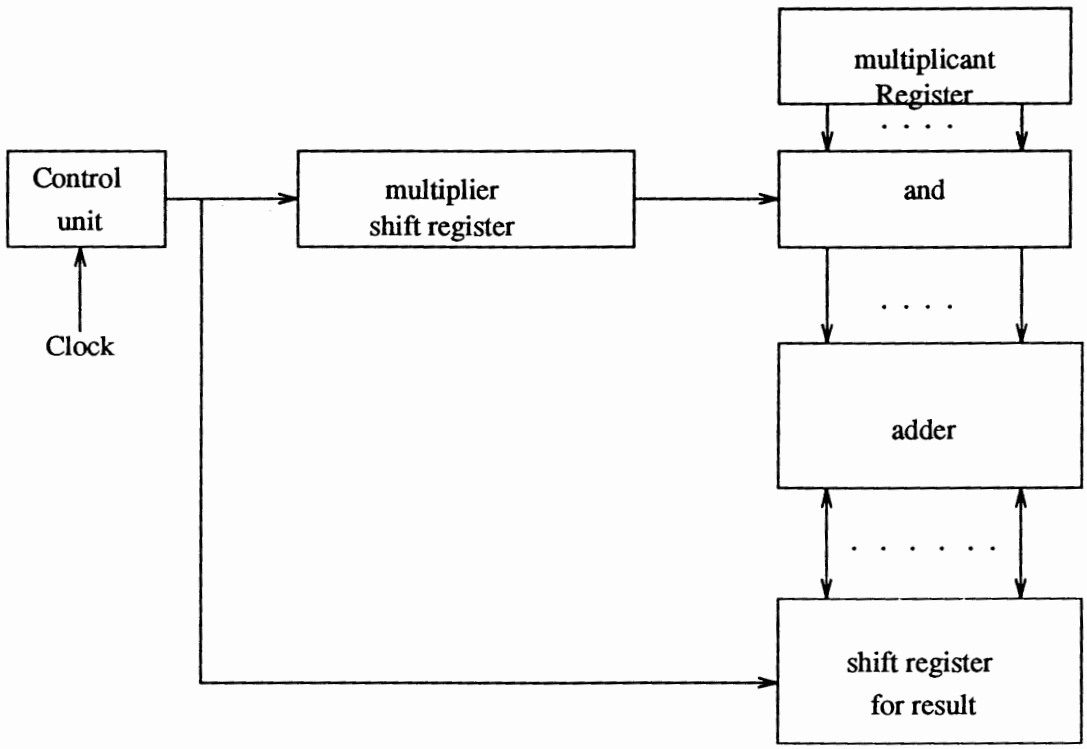

Figure 30. Multiplication operation.

This program is in the directory */u/palinisa/adl/compstr/compstr1*.

The program *mvhdl* bascially works in the same way as *svhdl* except for the subprograms *nali.c* and *pli.c*. The programs *nali.c* and *pli.c* implement all the operations in terms of gates. The program *nali.c* generates equations for each assignment

operations. If the program *nali.c* implements an addition operation *(2 (/:=/ e (plus a b)((8 9)))* as shown in example 8.2.3 then the program will create the equations shown below. These equations will be created as a component in **VHDL,** and this component can be called whenever it is needed.

d[0] = a[0] or b[0]

carr[0] = a[0] and b[0]

d[1] = a[1] or b[1] or carr[0]

carr[1] = (a[1] and b[1]) or ((a[1] and b[1]) and carr[0])

d[2] = a[2] or b[2] or carr[1]

carr[2] = (a[2] and b[2]) or ((a[2] and b[2]) and carr[1])

d[3] = a[3] or b[3] or carr[2]

carr[3] = (a[3] and b[3]) or ((a[3] and b[3]) and carr[2])

d[4] = carr[3]

The program *nali.c* creates different equations for different assignment statements and puts them in a component in **VHDL.** The component is called inside the main **VHDL** code. For subtraction operations, the program *nali.c* generates a 2's complement for the number to be subtracted and does addition. To implement a -b, the 2's complement of b is added to a. To implement the multiplication operation, the program *nali.c* creates a set of equations. If the program implements an multiplication operation shown in example 8.2.3, then the program will create the equations shown below. These equations will be created as a component as in addition operation.

d[0] = a[0] and b[0]

d[1] = (a[1] and b[0]) or (a[0] and b[1])

carr[1] = (a[1] and b[0]) and (a[0] and b[1])

d[2] = (a[2] and b[0]) or (a[1] and b[1]) or (b[2] and a[0]) or carr[1]

carr[2] = ((a[2] and b[0]) and (a[1] and b[1]) and (b[2] and a[0])) or (((a[2] and b[0]) or (a[1] and b[1]) or (b[2] and a[0])) and carr[1])

d[3] = (b[1] and a[2]) or (b[2] and a[2]) or carr[2]

carr[3] = ((b[1] and a[2]) and (b[2] and a[2])) or (((b[1] and a[2]) or (b[2] and a[2])) or carr[2])

d[4] = (b[2] and a[2]) or carr[3]

carr[4] = (b[2] and a[2]) and carr[3]

d[5] = carr[4]

Once all the operations are implemented it is included in the output **VHDL** file. The program *mvhdl* is in the directory */u/palinisa/adl/compst.*

Example -

STRUCT file

((

(input (a (p k1 3)) (b (p k1 3)) (c (p k1 3)) (d (p k1 3)))

(intern (e (p k1 3)) (f (p k1 6)) (g (p k1 3)))

(output (e (p k1 3)) (f (p k1 6)) (g (p k1 3)))

nil

)

(nodlista (5 m (/:=/) ((l 9) (1 8)) p k1 3) (6 m (/:=/) ((l 7) (1 6)) p k1 6)

(7 m nil ((1)) p k1 3) (4 m in nil p k1 3) (3 m in nil p k1 3)

(2 m in nil p k1 3) (1 m in nil p k1 3))

(coplista (9 s 1) (8 1 (0 5) (0 5) p k1) (7 s 1) (6 3 (0 5) (0 5) p k1))

(varlista (g 7) (f 6) (e 5) (d 4) (c 3) (b 2) (a 1))

(plisout)

(nalisimp (2 (/:=/ e (plus a b)((8 9)))

(3 (/:=/ f (c and d) ((6 7)))) br)

The corresponding **VHDL** description

```
entity clas is
port(
  a,b,c,d:in bit_vector(0 to 3);
  CLOCK:in bit;
  f:inout bit_vector(0 to 6);
  e,g:inout bit_vector(0 to 3));
end clas;

architecture adl1 of clas is
component addr port(
    a,c:inout bit_vector(0 to 3);
    clk: in bit;
    b:out bit_vector(0 to 3));
end component;
```

```
component mult port(
    a,c:inout bit_vector(0 to 3);
    clk: in bit;
    b:out bit_vector(0 to 6));
end component;

for all: ecou use entity work.ecou(acou);
for all: emult use entity work.emult(amult);

begin
process
begin
wait until CLOCK='1';
end process;

a1:addr
portmap ( a =>a, b => b, c => e,
        clk => CLOCK);


a2:mult
portmap ( a =>c, b => d, c => f,
        clk => CLOCK);


end adl1;
```

The example shown above is the output of the program svhdl. The output of the program mvhdl will be bascially the same except the change in the components. The

compnents will be in the gate level representation.

CHAPTER IX

CONCLUSIONS

In this thesis I have upgraded and modified the existing **DIADES** system so that it becomes a pre-processor for the commercially available **VHDL-** based synthesis/simulation system from Mentor Graphics Corporation. All the generated **VHDL** code is of Mentor Graphics **VHDL** format. All the generated **VHDL** code was compiled and simulated using the **VHDL** based Mentor tools. The results where discussed in the thesis.

The results in this thesis show that the **VHDL** code generated by the **VHDL-** based **DIADES** was complex and large compared to **ADL.** So the results prove that it is easier to program hardware using **ADL** than using **VHDL.** The results also show that it is difficult to describe parallel behavior of hardware using **VHDL.** The code generated by the **VHDL-** based **DIADES** for parallel behavior of hardware was very big and complex compared to the descriptions in **ADL.** So the results prove that it is very easy to use **ADL** to describe parallel behavior of hardware. In this way, by combining the existing **ADL-** based **DIADES** system and the features of **VHDL,** I have developed a new system in which an user can program hardware easily using **ADL.** The semantics of **ADL** language is similar to that of most of the programming languages, so that an user with little exposure to hardware description languages and with some experience with programming languages can program hardware using **VHDL-** based **DIADES.**

There are three paths a designer can chose from the **VHDL-** based **DIADES** system. The first one is through the output of the of the program gvhdl, which will be of behavioral design and this design is technology independent. The second second one is

through the output of the program svhdl, which will be of structural design and this design is technology dependent. The third one is through the output of the program mvhdl, which will be of data flow design and this design is also technology dependent. So by using **VHDL-** based **DIADES** system an user can select any of the three paths depending on the requirement.

The improving and modification for the datapath unit of the **DIADES** system is done and the improvement of the control unit of the **DIADES** system is left for the future research. The compiled file of a **VHDL** code can be simulated or can be used for implementing it in the **FPGAs.** The simulation is done in this thesis and the implementation of the design in **FPGAs** is left for future research. Extending the new approach presented for the parallelism problem for parallel descriptions with nested **fork** instructions is also left for future research.

The results prove that the combined through my work **DIADES/Mentor** system works correctly and is able to compile hardware from abstract parallel specification in **ADL** down to low level netlists. The results also prove that **ADL** allows for easy description of parallel and sequential behavior of hardware. To finally conclude, in this thesis a comprehensive and complete system has been developed that integrates the design on algorithmic level (**DIADES**), register transfer and logic level (Mentor Graphics tools). The system developed by me is not comparable to anything that currently exists in industry.

REFERENCES

[1] Dewey, A., VHDL and Next-Generation Design Automation, [*IEEE Design and Test of Computers June 1992.*]

[2] Dewey, A., and Geus, D., VHDL: Toward a Unified View of Design, [*IEEE Design and Test of Computers June 1992.*]

[3] Jayanta, R., Nanda, K., Rajiv, d., and Ranga, V., DSS: A Distributed High-level Synthesis System, [*IEEE Design and Test of Computers June 1992.*]

[4] Diades Research Group Report, [*PSU. 1989.*]

[5] DIADES User's Guide, [*PSU. 1989.*]

[6] Yang, L., Perkowski, M., and Smith, D., Design and Optimization of Microprogrammed Control Units in Diades, [*ISCAS 94*].

[7] Perkowski, M., A system for automatic design of digital systems, [*INFORMATICA 74*].

[8] ADL - source language of the system for automatic design, [*Organization of digital computers and microprogramming, Polish Scientific Publishers (PWN), 1976. Vol. 1, pp. 167-180.*]

[9] Smith, D., ADL a Behavioral Description Language, [*PSU. 1989.*]

[10] Perkowski, M., Parallel Programs in ADL and Their Semantics, Diades Research Group Report, [*PSU 1989.*]

[11] Smith, D., Description of GRAPH language, [*PSU. 1989.*]

[12] Liu, J., A Finite State Machine Synthesizer. [*The Thesis submitted in partial fullfillment of the requirement for the degree of MSC, Portand State Uni.,1989*]

[13] Mazor, S., and Langstraat, P., A Guide to VHDL, [*Kluwer Academic Publishers 1993.*]

[14] Carlson, S., Introduction to HDL-Based design using VHDL, [*Synopsys, CA, 1991.]*

[15] Bhasker, J., A VHDL primer, [*Prentice-Hall, 1992.]*

[16] Introduction to VHDL, [*Mentor Graphics, OR, 1993.*]

[17] Dillinger, T., A Logic Synthesis System for VHDL Design Description, [*IEE ICCAD-89.*].

[18] Kohavi, Z., Switching and Finite Automata Theory, [*McGraw-Hill, New York, 1978.*]

[19] Charles, H., Roth, Fundamentals of Logic Design [*West Publishing Company, 1985.*]

[20] Zissos, D., Logic Design Algorithms, [*Oxford University Press, 1972.*]